アスキー・ラーニングシステム　③応用コース

応用 C言語

三田典玄 著

改訂新版

アスキー出版局

アスキー・ラーニングシステム　③応用コース

応用C言語

三田典玄 著

改訂新版

アスキー出版局

商　標

- MS-DOS，Microsoft C Optimizing Compiler，Microsoft QuickC Compiler は，米 MICROSOFT 社の商標です．
- UNIX は，米 AT&T のベル研究所が開発し，AT&T がライセンスしています．
- Lattice C/DOS は，米 Lattice Inc.の商標です．
- LSI C-86 は，エル・エス・アイジャパン株式会社の商標です．
- TURBO C は，米 BORLAND 社の商標です．

そのほか，CPU 名，システム名などは一般に各開発メーカーの商標です．なお，本文中では TM，Ⓡマークは明記していません．

■イラスト　小田ヒロミ

はじめに

「応用C言語　改訂新版」をお贈りいたします．

この本は，前2巻の「入門C言語　改訂新版」，「実習C言語　改訂新版」に続く，C言語の応用について書かれた本です．

特にこの本では，前2巻での学習成果をもとにハードウェアの割り込み処理，UNIX／MS-DOS上でのプログラミングなど，現在のプログラマが直面しているさまざまなアプリケーションに対処できるようなテーマを選んで詳しく解説し，プログラム例も豊富に掲載しました．

本書では，C言語という枠組みの中でさまざまなアプリケーションを考える，というのではなく，C言語を窓口としてさまざまなコンピュータ応用の世界を広げていく，という視点に立ってすべての項目が書かれています．したがってC言語による応用プログラミングの本ではあるものの，ハードウェアの話，UNIXの話などコンピュータの本質的な知識を必要とする部分もあることをお断りしておきます．

「C言語の応用」については，この本で解説している事柄以外にも載せきれなかったさまざまなものがあることは承知の上で，通常のC言語プログラマの多くが直面するであろう問題はほぼカバーするよう項目を選んだつもりです．

この本を読んでいただくことによって，C言語のみならず，プログラムの本質と楽しみというものを少しでも理解していただければ，筆者としてこれに勝る喜びはありません．

なお，最後になりましたが，本書の執筆にあたってご協力をいただいた池田健二氏，民田雅人氏，伊藤義行氏，さらにページの都合で本書には掲載できませんでしたが，プログラムのご協力をいただいた小川史彦氏，数々の助言をいただいた真柄義弘氏に，深く感謝の意を表したいと思います．

1990年10月6日　三田典玄(nori@orange.orange.juice.or.jp)

C 言語ラーニングシステム　全 3 巻の構成

改訂版 C 言語のラーニングシステムは「入門 C 言語」,「実習 C 言語」,「応用 C 言語」の全 3 巻で構成されます.

従来の C 言語の書籍は，主にコンピュータをよく知っている人たちを対象に書かれており，数多くのことがらが 1 冊の本につまっていました．このため，基本的なコンピュータの知識を持っていない初心者が学習するには，かなり無理があったように思います．また逆に，コンピュータを使いこなしている上級者にとっては，細かい説明やくわしい内部構造など，ほんとうに知りたいことがいまひとつ載っていないことへの不満が少なからずあったことでしょう.

今回のシリーズは，初心者から無理なく学習でき，また上級者にも満足できるようにとの主旨から全体を 3 巻にわけて構成しました．以下に各巻の概要を紹介しておきます.

入門C言語：本巻では, C 言語で簡単なプログラムが書けるようになることをテーマとしています. C 言語を学ぶ上で不可欠なコンピュータの基本構造についても解説し，この先より大きなプログラムが書けるようになるための基礎的な力をつけることを主眼に置きました．最新の C コンパイラを使って説明していますので，初めての方でも安心して学習できます.

実習C言語：入門編で取り上げなかった「構造体」,「共用体」などの学習と，C 言語についてのすべての項目がわかるように体系的な整理を行っています．また，項目ごとに数多くの例題を挙げ，各自でプログラムを組んでいく際にマシンのそばに置いて，いつでも参照できるように構成しています. ANSI で追加された変数のクラスや関数宣言など最新の事項も，詳しく解説しています.

応用C言語：C 言語を本格的に使いこなすために，さらに高度なプログラミングと開発環境について学習します．とくにアセンブラとのリンク，日本語，ローカルエリアネットワークでのプログラミング，高速化について，UNIX 環境についてなどを，豊富なプログラム例とともに示しています．改訂にあたっては，多くのワークステーションで採用されている EUC コードの詳しい説明や，UNIX 上で利用できるデバッガ adb の解説を加えています.

本シリーズを読むにあたっては，実際にコンピュータを目の前に置いて動かすことを前提としていますが，書籍を読むだけでも体験的な学習ができるように配慮しています.

目　次

はじめに……3

C言語ラーニングシステム　全3巻の構成……4

第1章　本格的なプログラム開発の基礎知識――11

1.1　CPUのアーキテクチャに依存するプログラミング……13
■int型変数のビット長の違い……13
■ポインタ変数のビット長の違い……14
■レジスタ構成の違い……15

1.2　8086系CPUとポインタ変数の扱い……16
■8086系CPUのアドレスの表現方法……16
■メモリモデルの概念……18
■メモリモデルによるポインタ長の比較……20
■実アドレス領域を直接アクセスするプログラム……21

1.3　オペレーティングシステムに依存するプログラミング……27
■システムコールとは……27
■MS-DOSのシステムコールを使ったプログラム……28

1.4　移植性の高いプログラムの記述……32
■移植性の高いプログラムとは……32
■移植性の高いプログラムの書き方……32

1.5　プログラム記述のノウハウ……37
■プログラムの高速化……37
■バグの回避……39
■システム設計とプログラム開発ツール……40

第2章　オリジナルライブラリの作成 41

2.1　ライブラリとは……43

2.2　ライブラリ・マネージャの基本機能……45

2.3　ライブラリの保守／管理……48

■LIBコマンドの使い方……48

■UNIXでのライブラリ管理……49

2.4　実用ライブラリの作成……51

■割り込みサポートライブラリの作成……51

■割り込みサポートライブラリの利用……62

■ライブラリ作成の注意点……64

2.5　ヘッダーファイルによる簡易ライブラリの作成……65

■エスケープシーケンスとは……65

■エスケープシーケンス簡易ライブラリ……67

第3章　日本語処理 69

3.1　漢字コード体系……71

■7ビット系漢字コード……71

■8ビット系漢字コード……74

シフトJISコード……75

拡張UNIXコード……79

■コードの互換性……81

3.2　プログラムソース中の全角文字の扱い……82

■8ビットコードを使用できる処理系……82

■7ビットコードのみ使用できる処理系……83

3.3　全角文字の入出力……86

■全角文字の制御……86

■7ビット系コードの入出力処理……88

■8ビット系コードの入出力処理……90

3.4 全角文字列の編集 ……91
■シフトコードを使用することによる問題 ……91
■編集の前処理 ……92
■全角文字列編集用のライブラリ関数 ……97
■独自の内部処理コードの作成 ……100

3.5 サンプルプログラム —JFOLD— ……104

第4章 MS-DOS上でのプログラミング —— 119

4.1 システムコールを使ったプログラミング ……121
■intdos関数とは ……121
■データの受け渡し ……122
■ファンクションコールを呼び出すライブラリ関数 ……123
■内部割り込みを起動する関数 ……124
■現在のセグメントレジスタの値を得る関数 ……125
■intdos関数の必要性 ……125

4.2 ディレクトリを検索するシステムコール ……126
■ディレクトリの検索 —LDコマンド— ……126
■MS-DOSのバージョンによる動作の違い ……134
■指定されたパスの属性を得る ……135

4.3 子プロセスの実行 ……141
■子プロセスを起動する関数群 ……141
■system関数の使い方 ……145

4.4 サンプルプログラム —FINDF— ……149
■FINDFコマンドの概要 ……149
■プログラムの内部構造 ……152

第5章　割り込み処理――181

5.1 ハードウェアを直接扱うプログラム……183
■アセンブラの代わりとしてのC言語……183
■C言語でアプリケーションプログラムを組む……183

5.2 アセンブラとのリンク……184
■関数の呼び出し手順と変数の参照……184
■C言語プログラムとアセンブラプログラムの記述の違い……187
■インライン・アセンブル機能を持ったC言語の処理系……188

5.3 割り込み処理の概念……189
■割り込み処理とは?……189
■ハードウェア割り込みを利用したターミナルエミュレータ……190
■C言語での割り込み処理の扱い……192
■割り込み処理の記述……193
■必要なハードウェアの知識……194

5.4 割り込みプログラムの実際……197
■8086・CPUのハードウェア割り込みのメカニズム……197
■割り込みサービスルーチンの記述……199
■割り込みサービスルーチンのセットアップ……202

5.5 サンプルプログラム　―VTE―……210
■プログラムの概要……210
■main関数の概要……211
■送受信関数の概要……212
■画面制御関数の概要……214

第6章　UNIX上でのプログラミング――235

6.1　マルチタスク環境の考え方……237
■シングルタスクとマルチタスク……237
■マルチタスク環境の実現……237
■マルチタスク環境でのプログラミング……239
■UNIXのシステムコール……239

6.2　プロセス管理……240
■プロセスとはなにか……240
■プロセス管理のシステムコール……241

6.3　セマフォ(System V系, 4.3BSD)……246
■セマフォの概念……246
■「入場制限型」セマフォの利用……247
■「交通信号機型」セマフォの利用……247
■セマフォのシステムコール……248

6.4　プロセス間通信……254
■パイプの概念……254
■パイプのシステムコール……255
■メッセージの概念(System V系, 4.3BSD)……260
■メッセージのシステムコール……261
■共有メモリの概念(System V系, 4.3BSD)……268
■共有メモリのシステムコール……269

6.5　メモリ管理……270
■メモリ管理とUNIX……270
■malloc関数とその周辺関数……270
■ブレークバリューとbrk関数……271

6.6　割り込み処理……272
■割り込み処理の概念……272
■割り込み処理のシステムコール……273

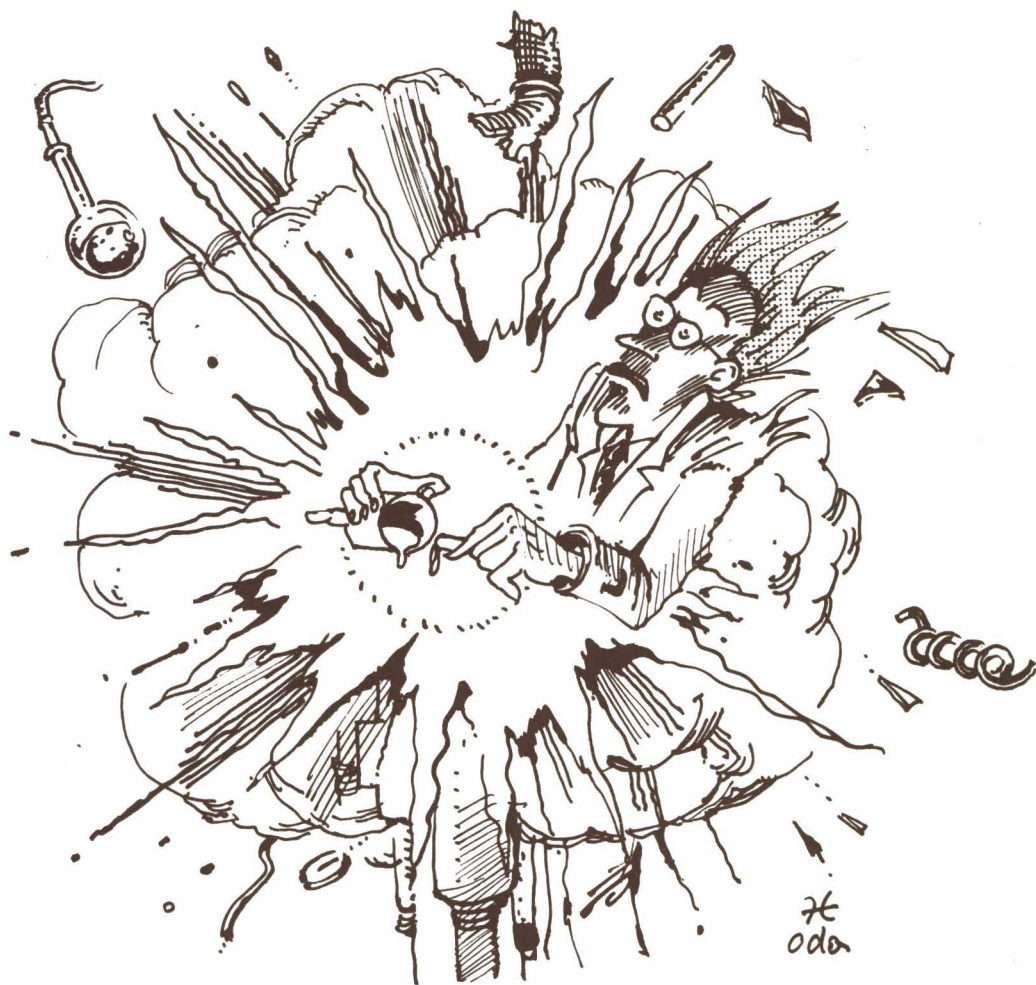
6.7 ソケット(4.3BSD) ……279
■クライアント/サーバーモデル ……279
■socketコールと名前づけ ……279
■socketコールの利用 ……280
■ソケットのシステムコール ……281
■サーバープログラムの解説 ……283
■クライアントプログラムの解説 ……285

第7章 プログラム開発環境 ―― 295

7.1 MAKE―コンパイル/リンク自動化ツール ……297
■MAKEの効用 ……297
■MS-DOSのMAKE ……300
■MAKEコマンドの動作の仕組み ……300
■MAKEコマンドの応用 ……302
■UNIXのmake ……303

7.2 DEBUG, SYMDEB―デバッグ用ツール ……308
■デバッガの機能 ……308
■最も基礎的なデバッグの実際 ……309
■デバッガの基本構造 ……313
■SYMDEBを使ったデバッグ ……314
■UNIX上でのデバッグ―adbの使い方 ……320
■高度なデバッガの問題点 ……323

索 引 ……324

第1章
本格的なプログラム開発の基礎知識

「入門C言語」,「実習C言語」では，特定のマシンやコンパイラに依存しない基礎的なプログラムを書くための知識を中心に解説してきました．しかし，実際のプログラム開発になると，C言語の知識のみではなかなか解決のつかない問題に直面します．とくにパソコン上のアプリケーションプログラムのように，画面の高速な制御や高度なユーザーインターフェイスを要求されるものになると，C言語自身の知識よりもそのコンピュータ・システムのことをよく知らないと実用に耐えるプログラムを作ることが困難になってきます．

ここでは，プログラム開発に使われることも多くなってきたパソコンでの開発を中心に，実用的なプログラムを組むうえでの基礎知識を整理してみましょう．

1.1　CPUのアーキテクチャに依存するプログラミング

C 言語は，アセンブラのようにきめ細かな CPU の制御が可能であることを本シリーズでは何度か述べてきました．しかしその反面，このことは CPU のアーキテクチャをよく理解していないと実用的なプログラムは書けないということでもあるのです．C 言語と CPU の関係でとくに問題となるのは，次のような点でしょう．

1. int 型変数のビット長の違い
2. ポインタ変数のビット長の違い
3. レジスタ構成の違い

そこで，以下にこれらの問題を整理してみます．

■ int型変数のビット長の違い

int 型変数のビット長は，一度に PUSH，POP できるレジスタのビット長が割り当てられますが，この値は CPU によってかなりまちまちです．**表 1-1** に代表的な CPU 上の int のビット長を示します．

	8080系CPU	6800系CPU	8086系CPU	68000系CPU	80386系CPU
CPU	8080 Z80 など	6800 6809 など	8088，8086 80188，80186 80286 など	68000 68010 68020 など	80386
intのビット長	16	16	16	32	32 MS-DOS では 16

表 1-1　代表的な CPU 上の int 型変数のビット長

68000 などのワークシテーションやミニコンで使われる CPU の場合，int は 32 ビット，すなわち long であり，8086 や 80286 などのパソコンで使われる CPU では 16 ビットになっています．このため，ビットフィールド*1などを使っているプログラムは，そのソースコードの一部を書き換える必要がでてきます．

---

*1　ビットフィールドは，int 型のみ有効

次の図1-1では，forによって変数iの値を増加させ，ループカウンタとして使おうとしていますが，intのビット長が16ビットであった場合は桁あふれを起こしてしまい，iが32768以上の値になるとプログラムの動きがおかしくなってしまいます．

図1-1　int型変数のビット長が問題となる例

■ ポインタ変数のビット長の違い

ポインタの操作はアドレスそのものを扱うこととは違いますが，その操作の対象になる数値は明らかにアドレスそのものです．ここで，アドレス値をintなどの別の整数値に変換して使用すると多くの問題が起こります．

とくにバークレー系をはじめとするUNIX上のC言語のコンパイラは，intもポインタもともにlong(32ビット)であるため，ポインタ型とint型をいい加減に扱ったプログラムでもよく動きます．しかし，MS-DOSや16ビット系CPUのC言語では，これらの型をきびしくチェックしないと，そのままでは動かないものがかなりあります．とくにUNIX上のソースリストでは，関数の返値(戻り値)を明示的に宣言していないことも多く，この場合はintが返ってくるものと仮定されます．しかし，intのビット長がUNIXとMS-DOS上のC言語では異なるので，問題が起こってしまいます．

UNIX上のC言語で書かれたソースリストをMS-DOS上に移植する場合などは，とくにこの点を注意して見てください．

次ページの図1-2では，ポインタ変数のポインタ値(アドレス)をint型の変数に保存していますが，もし，intのビット長が16ビットで，ポインタ変数のビット長が32ビットであった場合は，ポインタ値(pの値)が正しく保存されません．こういった問題は，MS-DOS上のCコンパイラでラージモデルを使用したときに起こります(ラージモデルについては次節を参照)．

複雑なポインタの演算を行いたい場合などに，このようなことをやりたい衝動にかられますが，やはりこういったコーディングは控えるべきでしょう．

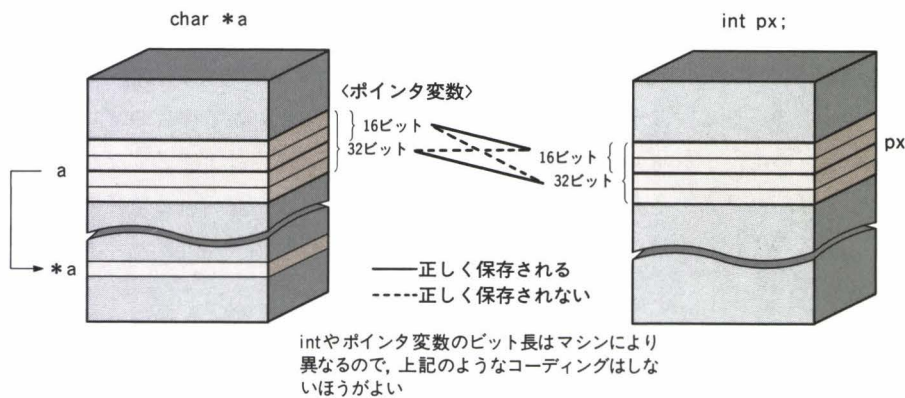



図1-2　ポインタ変数とint型変数の混同

■ レジスタ構成の違い

CPUはその各々のアーキテクチャの違いによって，レジスタの構成がかなり違います．このためレジスタ変数を使っている場合，プログラムリストではレジスタ変数を割り当てているのに，実際にはコンパイラが自動的にメモリ上に割りつけていたということもあります．レジスタ変数は，8086系CPUでは2つ(通常はSIレジスタとDIレジスタが使われる)，68000系CPUでは7～8程度使用することが可能です．

こういったレジスタ変数の扱いは，アセンブラとのインターフェイスをとるときに注意しておく必要があります．とくにアセンブラとリンクするプログラムを作成する場合は，レジスタ変数はなるべく使わない方がよいでしょう．また，レジスタのビット長も意識しておかないと，アセンブラプログラムのコーディング時に問題となります．

1.2　8086系CPUとポインタ変数の扱い

多くの16ビット・パーソナルコンピュータで採用されている8086系CPUでは，ポインタの問題はさらに複雑になってきます．そこでまず，8086系CPUのアドレスの表現方法と，C言語でプログラムを作成する際に問題となる「メモリモデル」についてくわしく解説します．

■ 8086系CPUのアドレスの表現方法

8086系CPUでのプログラミングの際，かならず問題となるのがメモリモデルです．これは，8086系CPUのアドレス表現が**セグメント**と呼ばれる論理空間で分割されているために起こる問題です*2．もともと，セグメントの概念はマルチタスクのオペレーティングシステムで複数のプロセスを管理するために設けられたもので，シングルタスクのオペレーティングシステムでは必要なかったものです．

8086系CPUでは，セグメントを「アドレス表現の上位の桁のことである」と単純に理解してもかまいません．アドレスの下位の桁は**オフセット**と呼ばれ，このオフセットとセグメントでメモリ上のアドレスを決定します．

こういったセグメントとオフセットで表現されるアドレス表現のことを**論理アドレス**といいます．これはあくまで論理的なアドレスの指定方法であり，実際にメモリ上に割りつけられるアドレス(**物理アドレス**という)を表すものではありません．実際の物理アドレスは，オフセットとセグメントを加算した値になります．しかし，この加算はただ単に2つの値を足しただけでは得られません．簡単にいうと，セグメント値を4ビット上位にずらして20ビットの値とし，オフセット値を加えたものが実際の物理アドレスになります．くわしい計算方法は，次ページの**図1-3**を見てください．

8086系CPU(あるいは仮想8086モード)は物理アドレス長を20ビットしか持っていないのですが(すなわち1Mバイトのアドレス空間を扱える)，このアドレス空間を表すためのセグメントとオフセットのレジスタ長がそれぞれ16ビットであるために，こういった面倒な計算をしなくてはならないのです．

なお，参考までに8086CPUの全レジスタを**図1-4**に示しておきます．C言語だけでプログラムを作成する際には(レジスタ変数を扱わない場合)，レジスタを意識する必要はまずありません．しかし，システムコールを使ったプログラム(第4章で解説)やアセンブラとリンクするプログラム(第5章で解説)，およびデバッガの利用(第7章で解説)の際には，各レジスタとその役割を知っておかなければなりません．

---

*2　セグメントの概念については，「はじめて読む8086」(蒲地輝尚著　アスキー出版局発行)を参照するとよい．

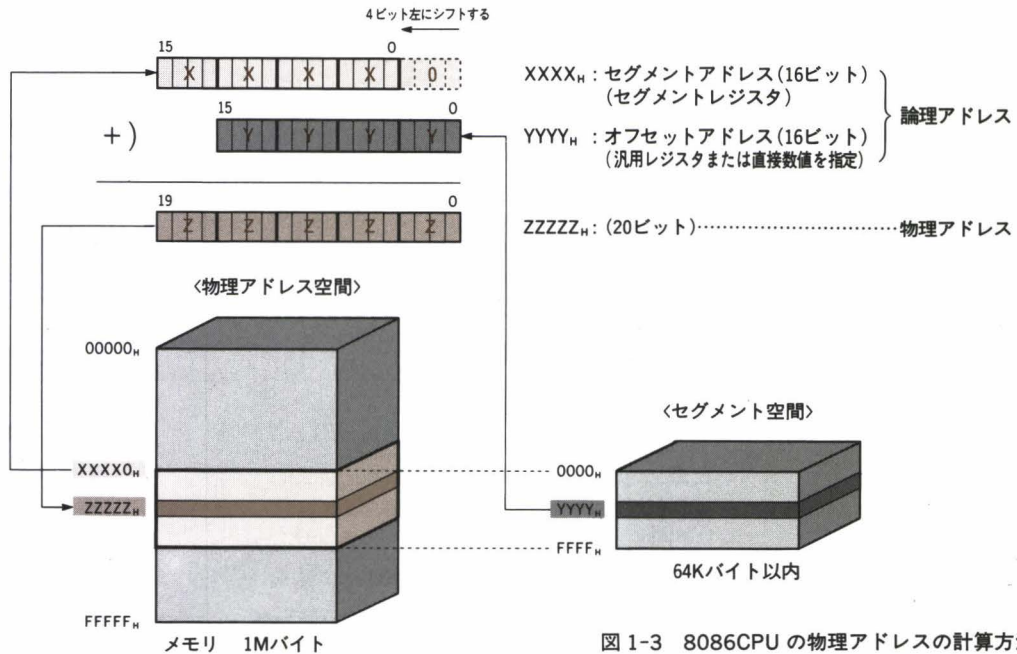



図1-3　8086CPUの物理アドレスの計算方法

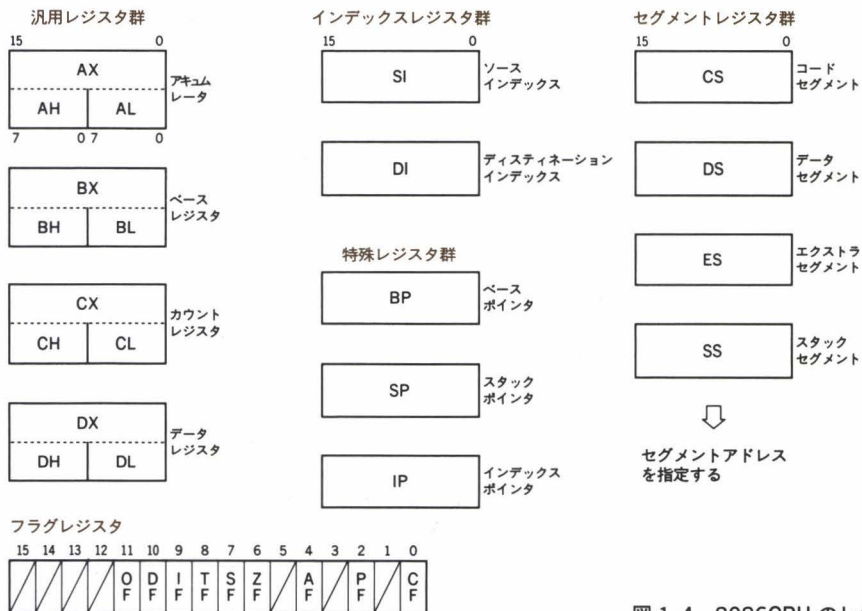
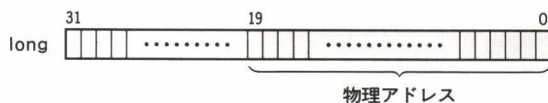



図1-4　8086CPUのレジスタ構成

Cコンパイラには，これらの物理アドレスをポインタとして直接渡す(アドレスは20ビットで表されるものとしlongの型を持つ)ものと，論理アドレスでのみ使えるもの(アドレスは32ビットのlong型になる)とがあります(**図1-5**)．Microsoft C(以後はMS-Cと記述)やTurbo Cなど最近の処理系は後者のようにポインタが扱われます．

〈物理アドレスをポインタとして直接渡す場合〉

31　19　0
long
物理アドレス

〈論理アドレスをポインタとして渡す場合〉

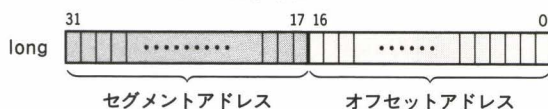



図1-5　Cコンパイラによるポインタ変数の扱いの違い

次項で解説するコンパイラのメモリモデルによっては，ポインタ演算をオフセットの部分に限っているものがあります．この場合，プログラムの実行速度が高速になり，コンパイルされたプログラムサイズも小さくなりますが，セグメント境界での問題(オフセットを計算して桁上がりが起こった場合の問題)を常に考えていないと，大きなデータを扱った場合などにとくに問題が発生することがあります．

一方，そのような問題を解決するためにポインタ演算をライブラリ関数で行っているメモリモデルが提供されている処理系もあります．この場合，セグメント境界での問題がなく，ポインタの演算は8086･CPUの特性を考えずにすみますが，その反面，ポインタ演算のたびにセグメントとオフセットの計算をしているわけで，実行速度が問題となるアプリケーション･プログラムでは注意が必要です．

■ メモリモデルの概念

メモリモデルは8086･CPUのメモリの使い方を，いくつかの典型的な場合に分けたものです．コンピュータのなかのデータ領域は，プログラムの入る**コード領域**とデータの入る**データ領域**に大きく分かれています．また，このほかにスタックを使うための**スタック領域**もあります．メモリモデルはこれらのセグメントの扱いと，それをどのようにデータ／コードのセグメント空間に分けるかということを指定します．

代表的なメモリモデルを次に見ていきましょう．

スモールモデル(Small Model)

コード領域，データ領域をそれぞれ1つずつのセグメント内に収めます．プログラムの実行中にセグメント値が変わることはありません．すなわち，コード，データともポインタのビット長は16ビットとなります．これはコードは64Kバイト以内，データも64Kバイト以内ということを表しています．

ミディアムモデル(Medium Model)

コード領域のセグメントが実行中に常に変わります．つまり，プログラム(C言語では関数)へのポインタのビット長は32ビットになります．データへのポインタのビット長は16ビットです．もちろん，実行中にデータへのポインタのセグメント値が変わることはありません．

ラージモデル(Large Model)

データ領域のセグメントが実行中に常に変わります．つまり，データ(C言語では変数)へのポインタのビット長は32ビットになります．プログラムへのポインタのビット長も32ビットです．もちろん，実行中にこのポインタのセグメント値は変わることになります．

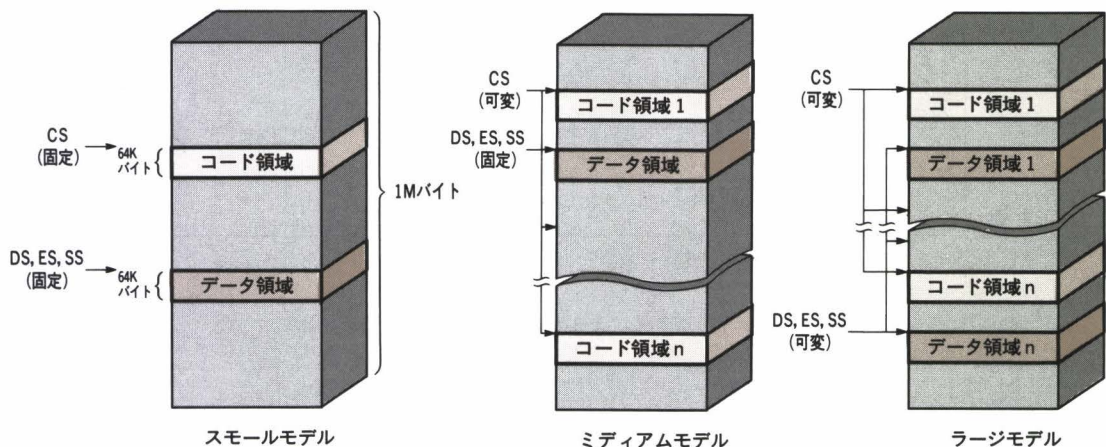



図1-6　メモリモデルによるコード領域／データ領域の相違

以上が代表的なメモリモデルですが，最近では**コンパクトモデル**(ミディアムモデルとは逆にコードセグメント値を固定し，データセグメント値を可変にしたもの)など，いろいろなメモリモデルが考えられています．

このなかには，処理系によってその名前が違ったり，コンパイラの動きが違うこともあります．たとえばMS-Cでは**ヒュージ**(huge)というメモリモデルをサポートしています．このヒュージモデルはデータセグメントの値をコンパイラのコードが常に管理し，64Kバイト以上の大きさの1つのデータの〝かたまり〟(たとえば1つの配列)をアクセスできるようにしたものです．ラージモデルとそのアセンブラレベルでのデータの扱いは同様ですが，考え方が異なります．つまり，ヒュージモデルは特定のコンパイラがサポートするメモリモデルということになりますから，厳密な意味でのメモリモデルとは少し違います．

各社のCコンパイラのメモリモデルは，26ページの表1-3にまとめていますから，参照してください．

■ メモリモデルによるポインタ長の比較

8086・CPUでは，その上で動くC言語のポインタのビット長がプログラミングをするうえで重要な要素になってきます．「char ＊data」の変数のビット長がメモリモデルや，その対象が関数か変数かによって変わってしまうのですから，これは大きな問題です．そこで，以下に代表的なメモリモデルと，そのメモリモデルでのポインタのビット長を比較してみました(**表1-2**)．

宣言 ＼ メモリモデル		スモール	ミディアム	コンパクト	ラージ
データへのポインタ	〈type〉 ＊	16	16	32	32
	〈type〉 near ＊	16	16	16	16
	〈type〉 far ＊	32	32	32	32
関数へのポインタ	〈type〉(＊)()	16	32	16	32
	〈type〉(near ＊)()	16	16	16	16
	〈type〉(far ＊)()	32	32	32	32

〈type〉………char, int, long, float, doubleなど

表1-2　メモリモデルによるポインタ変数のビット長の比較

この表でFARはFARポインタ，すなわちビット長が32ビットであるポインタのことをいい，NEARは16ビット長のポインタのことをいいます[*3]．

＊3　FARとNEARについての詳細は，8086のアセンブラに関する書籍を参照のこと．

大きなプログラムを組む場合やアセンブラとのインターフェイスが問題となる場合，さらにハードウェアの直接のアクセスをともなう場合は，ラージモデルを使う方がプログラムを楽に組めます．また逆に，プログラムの規模が小さく，扱うデータも少なく，またハードウェアを直接扱うことのない場合は，スモールモデルでプログラムを組む方がよいでしょう．

そのほかのメモリモデルはそれぞれの状況に応じて選ぶようにしましょう．

■ 実アドレス領域を直接アクセスするプログラム

ポインタ操作のプログラム例を以下に示します．ここでは PC-9801 シリーズ(XA, LT などは除く)のテキスト画面に直接文字を書くプログラムを作ってみましょう．

PC-9801 シリーズでは，テキスト画面はビデオ RAM(V-RAM)と呼ばれるメモリ上の特定のアドレスに文字コードとその属性を書き込むことによって，ハードウェアがその領域を読み込んで文字を画面に表示します．

画面に文字を書くというプログラムは，MS-DOS のシステムコール(次節参照)や C 言語の printf 関数でも行えますが，これらの方法ではすばやく画面に書くことができません．画面をより速く表示したい場合は，以下の C 言語のプログラムを使って 1 文字ずつ画面に表示した方が速いのです．

これは printf 関数では，

printf 関数 → MS-DOS の文字出力システムコール → PC-9801 の ROM 内ルーチン → V-RAM

という順序で画面に書かれるべきデータが流れるので「たいへん時間がかかる」のに対して，ここで紹介する C 言語の関数では，このプログラムが直接 V-RAM に文字データを書いているからです(もちろん，そのために移植性が犠牲になっているのですが…)．

PC-9801 の V-RAM エリアは CPU の実アドレスで A0000h 番地から始まり，その色や属性を決定するエリアは A2000h 番地から始まります．次ページの**図 1-7** に同シリーズのメモリマップを示しておきます．他機種の方もマニュアルにはこういったメモリマップが掲載されていますから，テキスト V-RAM の先頭アドレスを調べてください．

リスト 1-1 のプログラムでは，指定された 1 文字を指定された属性で指定された位置に書き込みます．

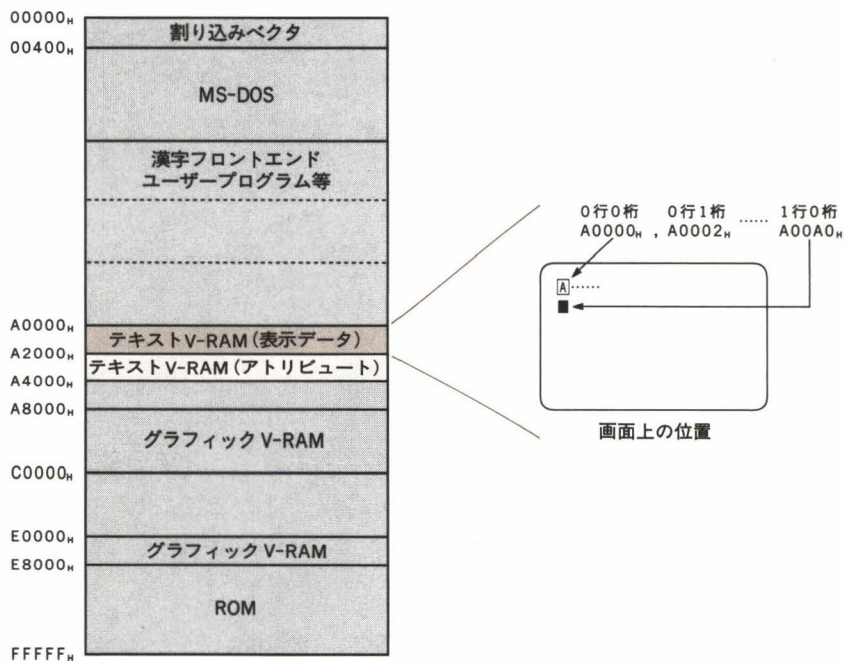



図1-7　PC-9801シリーズのメモリマップ

```
 1: /*
 2:      Display 1-characters for TEXT-RAM Directory
 3: */
 4:
                  行の値(0～24)
 5:
 6: void    dspk1(int law, int col, unsigned int moji, int color_f,
    int color_m)
                               桁の値(0～79)
 7:
 8:     /* color_f */
 9:     /* 文字色 (0 to 7)   */
10:     /* 0 ..... 黒        */
11:     /* 1 ..... 青        */
12:     /* 2 ..... 赤        */
13:     /* 3 ..... 紫        */
14:     /* 4 ..... 緑        */
15:     /* 5 ..... 水色      */
16:     /* 6 ..... 黄        */
17:     /* 7 ..... 白        */
18:
19:     /* color_m */
20:     /* 文字モード (0 to 7)                   */
```

```
21:      /* 0 ..... Normal                        */
22:      /* 1 ..... Brink                         */
23:      /* 2 ..... Reverse                       */
24:      /* 3 ..... Brink & Reverse               */
25:      /* 4 ..... Under-Line                    */
26:      /* 5 ..... Under-Line & Brink            */
27:      /* 6 ..... Under-Line & Reverse          */
28:      /* 7 ..... Under-Line,Reverse & Brink */
29:
30:      {
31:      unsigned long   adr_tmp;
32:      int    adr_val;
33:      char   far *adr_ch00;
34:      char   far *adr_ch01;
35:      char   far *adr_ch02;
36:      char   far *adr_ch03;    } 実アドレスのポインタ変数
37:      char   far *adr_at00;
38:      char   far *adr_at01;
39:      char   far *adr_at02;
40:      char   far *adr_at03;
41:
42:                                               オフセットアドレスの算出
43:      adr_tmp = 160L * (unsigned long)law + 2L * (unsigned long)col;
44:                           セグメントアドレス    オフセットアドレス
45:      adr_ch00 = (char far *)(0xA0000000L + adr_tmp);
46:      adr_at00 = (char far *)(0xA2000000L + adr_tmp);  } 書き込むアドレスの算出
47:
48:      adr_val = (color_f << 5) | 0x01 | (color_m << 1);
49:                                               文字色／文字モードの決定
50:      if(moji < 0x0100)……漢字コードでない場合の処理
51:          {
52:          adr_ch01 = adr_ch00;
53:          adr_ch01++;
54:          adr_at01 = adr_at00;
55:          adr_at01++;
56:          *adr_ch00 = moji;
57:          *adr_ch01 = 0;
58:          *adr_at00 = adr_val;
59:          *adr_at01 = 0xFF;
60:          }
61:      else ……漢字コードの場合の処理
62:          {
63:          moji = sjis2jis( moji );……シフトJIS→JIS変換
64:          adr_ch01 = adr_ch00 + 1;
65:          adr_ch02 = adr_ch00 + 2;
66:          adr_ch03 = adr_ch00 + 3;
67:          adr_at01 = adr_at00 + 1;
68:          adr_at02 = adr_at00 + 2;
69:          adr_at03 = adr_at00 + 3;
70:
71:          *adr_ch00 = *adr_ch02 = (moji >> 8) - ' ';
72:          *adr_ch01 = moji;
73:          *adr_ch03 = moji | 0x80;
74:
75:          *adr_at01 = *adr_at03 = 0xFF;
76:          *adr_at00 = *adr_at02 = adr_val;
```

```
77:         }
78:     }
79:
80:
81: sjis2jis(int c)……………シフトJIS→JIS変換ルーチン（第3章を参照）
82: {
83:     int hi, lo;
84:
85:     hi = (c >> 8) & 0xff;
86:     lo = c & 0xff;
87:     hi -= (hi <= 0x9f) ? 0x71 : 0xb1;
88:     hi = hi * 2 + 1;
89:     if (lo > 0x7f)
90:         lo--;
91:     if (lo >= 0x9e) {
92:         lo -= 0x7d;
93:         hi++;
94:     } else
95:         lo -= 0x1f;
96:     return hi << 8 | lo;
97: }
```

リスト 1-1　DISP.C

このプログラムは MS-C(または Turbo C)を使うことを前提に書かれています．ポインタの操作がそのままの「セグメント：オフセット」形式の操作であるからです(45 行目，46 行目)．

以上の関数を使うためのヘッダーファイルも作っておきましょう．

```
 1: /*
 2:         Headers for
 3:         Display 1-characters for TEXT-RAM Directory
 4: */
 5:
 6:
 7: #define         C_BALCK         0
 8: #define         C_BLUE          1
 9: #define         C_RED           2
10: #define         C_MAGENTA       3
11: #define         C_GREEN         4
12: #define         C_SKYBLUE       5
13: #define         C_YELLOW        6
14: #define         C_WHITE         7
15:
16: #define         M_BLINK         (1)
17: #define         M_REVERSE       (2)
18: #define         M_UNDERLINE     (4)
19: #define         M_NORMAL        (0)
20:
```

```
21: #ifdef  LINT_ARGS
22: void    dspk1(unsigned , unsigned , unsigned , unsigned , unsigned)
23: #else   /* LINT_ARGS */
24: void    dspk1();
25: #endif  /* LINT_ARGS */
26:
27: #define         NORMAL          C_WHITE,M_NORMAL
```

リスト 1-2　DISP.H

このプログラムを使って，画面の左上に「A」の文字を緑色で書いてみましょう．属性はブリンクとアンダーラインとします．

```
 1: /*
 2:     TEXT RAM Direct-Access procedure TEST.
 3: */
 4:
 5: #include    <stdio.h>
 6: #include    "disp.h"
 7:
 8: main()
 9: {
10: dspk1(0,0,(unsigned int)('A'),C_GREEN,M_BLINK | M_UNDERLINE);
11: }
```

リスト 1-3　TEST1.C

このプログラムをコンパイルします．以下にコンパイル用のバッチファイルを示します．

```
    ┌─ -AS(スモール)，-AM(ミディアム)でも可
CL -AL -Gs -Od -Zde -FeDISPTEST DISP.C TEST1.C
        └─ 指定しなくてもよい
```

リスト 1-4　TESTD.BAT

このプログラムを実行すると，速い速度で画面表示ができることがわかるでしょう．このプログラムはラージモデルでコンパイルしていますが，FARポインタを指定しているため，MS-Cの「-Zde」オプションをつけてコンパイルしています(MS-C Ver4.0以降ではデフォルトで指定されている)．つまり，ラージモデル以外のメモリモデルでコンパイルしても同じように動くわけです．また，逆にラージモデルでコンパイルすることが前提であれば，FARというキーワードを取り去って，「-Zde」オプションをつけずにコンパイルしても同じ結果が得られます．

以下に，MS-DOS版の主要なCコンパイラのメモリモデル指定オプションを示します．前述したように，各メモリモデルの特徴を念頭に入れたうえで，プログラムを作成しなければなりません．どんなメモリモデルを選択するかによってプログラムの構造や記述が変わってくることもありますから，メモリモデルはプログラム設計時にきちんと決めておくようにしましょう．

表1-3を見ればわかるとおり，最近ではどのコンパイラでもほぼ同じメモリモデルが用意されています．このなかで，Turbo Cについている**タイニィモデル**は，コードとデータのサイズを64Kバイト以内に収めることによって，「.COM」形式の実行ファイルを作成するためのものです*4.

メモリモデル	コードサイズ	データサイズ	MS-C	Quick C	Turbo C	LSIC-86	Lattice C
Tiny	64Kbyte		-AT †2	——	-mt	——	——
Small	64Kbyte	64Kbyte	-AS	-AS	-ms	-ms	-ms
Mediaum	1Mbyte	64Kbyte	-AM	-AM	-mm	-mp	-mp
Compact	64Kbyte	1Mbyte	-AC	-AC	-mc	-md	-md
Large	1Mbyte	1Mbyte †1	-AL	-AL	-ml	-ml	-ml
Huge	1Mbyte	1Mbyte	-AH	-AH	-mh	——	-mh

†1 ：データ配列などの1つの領域の大きさが64Kbyteを超えることはできない
†2 ：MS-C Ver.6.0 以降でサポート

表1-3　Cコンパイラ別のメモリモデル指定オプション

＊4　MS-C Ver6.0でもサポートされている

1.3 オペレーティングシステムに依存するプログラミング

オペレーティングシステムというと，MS-DOSとかUNIXといった名前がすぐに挙げられるでしょう．ところで，これらの持っている「dir」とか「ls」といったコマンドを，オペレーティングシステムそのものと勘違いしている人もいるようです．これらのコマンドはオペレーティングシステムそのものではなく，あくまでオペレーティングシステムの持っているコマンド(ユーザーインターフェイス)であり，本来の意味でのオペレーティングシステムとは違います．

オペレーティングシステムの実体は，プログラマのレベルから見ると，こういったコマンドよりもアプリケーションプログラムに供給されるサービスルーチンにあるのです．ここでは，これらのサービスルーチンを利用したプログラムの記述について解説します．

■ システムコールとは

オペレーティングシステムにある機能で，アプリケーションから使用可能なサービスルーチンを**システムコール**と呼びます．オペレーティングシステムは多くの場合，メモリ上に常駐していますから，アプリケーションはいつでもこれらのサービスルーチンを使うことができます．

UNIXのCコンパイラでは，これらのサービスルーチンはライブラリ関数によってサポートされていますが，パソコン用のBASIC言語などの高級言語ではこれらのシステムコールを直接呼び出す命令は用意されていないので，アセンブラでシステムコールを呼ぶルーチンを作り，それを高級言語の側から呼び出して利用するのが一般的でした．

これに対して，現在市販されている多くのMS-DOS上のCコンパイラでは，これらのシステムコールを使うライブラリがあらかじめ用意されているのが普通です．ただし，ディスクのディレクトリ領域へのアクセスなどいくつかのシステムコールはライブラリ関数でサポートされていないので，ソフトウェア割り込みを用いてシステムコールを直接呼び出す必要があります(とくにディレクトリ操作については，第4章でくわしく解説している)．

また，これらのライブラリの中身はどのCコンパイラでもほぼ同じなので，このシステムコールを使うライブラリを利用しているC言語のソースリストは，MS-DOSの上で動くかぎりかなりの互換性を持っています．

次ページの図1-8では，システムコールの概念を図にしてみました．

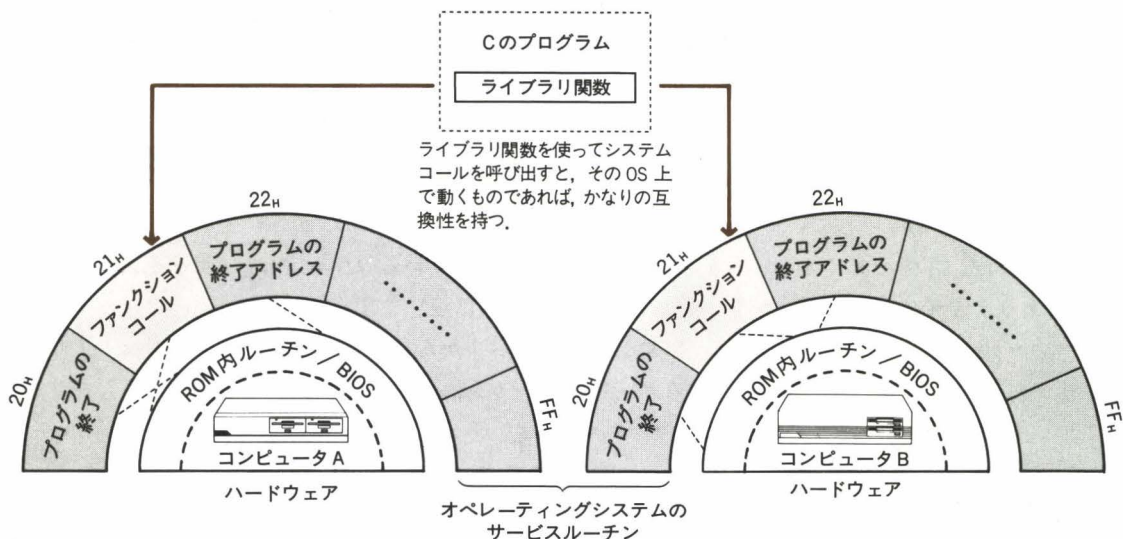



図1-8　システムコールの概念

次項では，MS-DOS上でのプログラミングを例にとり，システムコールを使ったプログラム例を見てみましょう．

■ MS-DOSのシステムコールを使ったプログラム

MS-DOSでは，システムコールを使うことによって，さまざまなハードウェアへのアクセスが行えるようになっています．コマンドラインの解析や現在のシステムの状態の把握などもサポートされています．もちろん，システムの内蔵時計へのデータのセットや現在時間の取得なども，これらシステムコールの機能のうちです．

システムコールをアプリケーションから使うには，機械語命令の「INT」(INTerrupt：割り込み)を使います．INT命令を発生するライブラリ関数としては次のものが代表的です(各関数の詳細については第4章を参照のこと)．

int86()，int86x()，intdos()，intdosx()，bdos()

これらのライブラリを使うには，ヘッダーとしてDOS.Hをインクルードします．このDOS.Hには，レジスタ構造体という構造体が定義してあります．この構造体にINT命令を実行するまえに必要なレジスタへの値をセットします．また，INTから戻ってきたときも，この構造体に値がセットされて返ってきます．

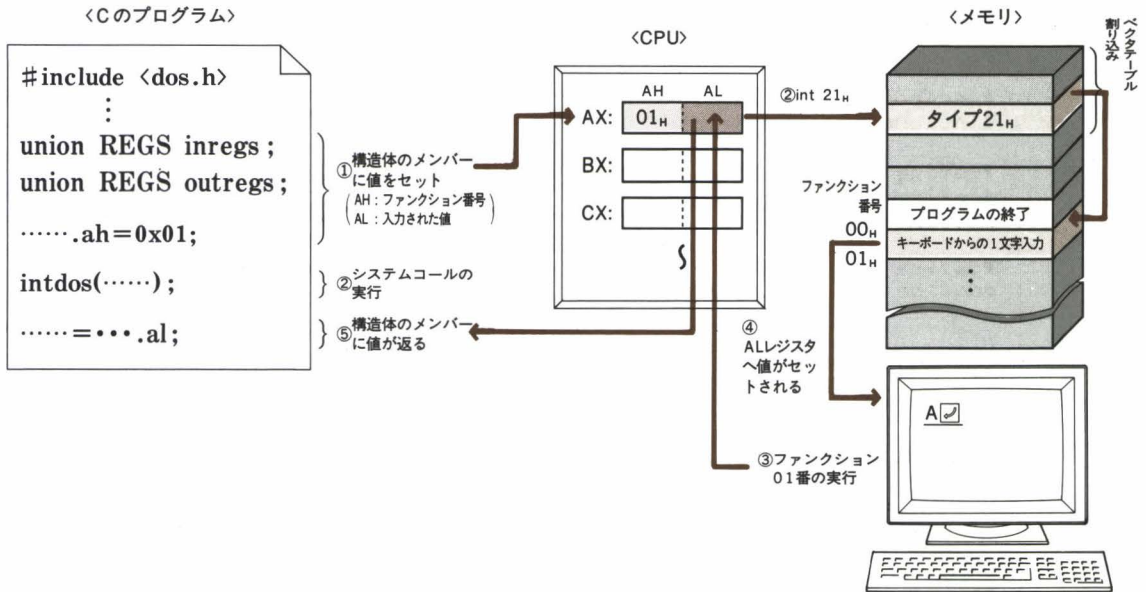



図 1-9 システムコールの仕組み

次ページの**リスト 1-5** に MS-DOS のシステムコールを使ったプログラムを示します．ここでは，MS-DOS のシステムコールの「0Bh」を使って，キーボードからの入力をチェックするプログラムを作ってみましょう*5.

このプログラムは MS-C などでは kbhit 関数というライブラリ関数で実現できますが，ここでは MS-DOS のシステムコールを使って実現してみます(kbhit 関数もその内部では同じシステムコールを使っている).

*5 MS-DOS のシステムコールについては，「MS-DOS3.1 ハンドブック」(アスキー書籍編集部編著 アスキー出版局発行)などの書籍を参照のこと.

```
 1: /*
 2:      Keyboard Hit ? or Not ?
 3: */
 4:
 5: #include     <stdio.h>
 6: #include     <dos.h> ......................MS-DOSインターフェイス・ヘッダーファイル
 7:
 8: #define      BOOL         int
 9: #define      FALSE        0
10: #define      TRUE         1
11:
12: #define      INT_DOS      0x21
13: #define      CHECKBUFF    0x0B
14:
15: /*
16:      Keyboard buffer check ......................キーボードが押されたかどうかを調べる関数
17: */
18: BOOL    khit()
19: {
20: union   REGS    inpreg; ......................入力レジスタ共用体
21: union   REGS    outreg; ......................出力レジスタ共用体
22:
23: inpreg.h.ah = (unsigned char)CHECKBUFF; ......................AHレジスタに値をセット
24:
25: int86(INT_DOS,&inpreg,&outreg); ......................int命令を実行
26:
27: if((unsigned char)0 == outreg.h.al) return(FALSE); ┐
28: else                                return(TRUE);  ┘ 戻り値の評価
29: }
30:
31: /*
32:      Main procedure
33: */
34: main()
35: {
36: printf("¥nWait for Hit Keyboard..");
37: while(!khit()); ......................キーが押されるまで待つ
38: printf("¥a¥rHit!                          "); getch();
39: }
```

リスト1-5　KBH.C

ここでレジスタの値を受け渡す構造体は，構造体と構造体の共用体になっています．型はREGSというタグ名でDOS.Hに定義されており，ワードアクセス，バイトアクセスがともに可能です．また，汎用レジスタだけでなく，セグメントレジスタもMS-DOSのシステムコールでは使っていますので，これらもSREGSというタグ名で用意されています．次にMS-CのDOS.Hの中身を示します．

```
 1: struct WORDREGS {
 2:     unsigned int ax;
 3:     unsigned int bx;
 4:     unsigned int cx;
 5:     unsigned int dx;
 6:     unsigned int si;
 7:     unsigned int di;
 8:     unsigned int cflag;
 9:     };
10:
11: struct BYTEREGS {
12:     unsigned char al, ah;
13:     unsigned char bl, bh;
14:     unsigned char cl, ch;
15:     unsigned char dl, dh;
16:     };
17:
18: union REGS {
19:     struct WORDREGS x;
20:     struct BYTEREGS h;
21:     };
22:
23: struct SREGS {
24:     unsigned int es;
25:     unsigned int cs;
26:     unsigned int ss;
27:     unsigned int ds;
28:     };
```


図 1-10　REGS 共用体／SREGS 構造体(MS-C)

これらのレジスタ共用体は，各社のC言語によって違いがあります．**図 1-10** では MS-C の例を示しましたが，ほかの処理系をお使いの方は，一度 DOS.H ヘッダーファイルを見てからプログラミングを行うべきでしょう．

よりきめの細かい DOS の制御やディレクトリエントリを扱うプログラムなどは，こういった DOS 関係のシステムコールを使わざるをえません．第 4 章にこの例の 1 つとして，UNIX のディレクトリ汎用検索コマンドである find とほぼコンパチブルな MS-DOS 版のプログラムを用意しましたので参考にしてください．

いずれにしても，システムコールを使ったプログラムでは，CPU のレジスタ構成などのアーキテクチャや，オペレーティングシステムの性質をよく理解したうえでのプログラミングが必要になります．

1.4　移植性の高いプログラムの記述

C言語でのプログラミングに限らず，大切なことはなるべく機種に依存しないプログラムを書くということでしょう．とくにC言語では，その移植性のよさをフルに使ったプログラムが期待されているのではないでしょうか？　1.2節では，8086CPU上のプログラムとして，V-RAMをアクセスする例を取り上げましたが，このようなハードウェアを直接扱う関数は，ハードウェアに依存しない関数と切り分けるなどの工夫が必要です．

この節では，移植性の高いプログラムの記述について考えてみましょう．

■ 移植性の高いプログラムとは

C言語でプログラムを作る場合，移植性の高いプログラムとは以下のように定義できるでしょう．

①変数の型の定義がきちんと意識されているプログラム

②機種に依存する部分が機種に依存しない部分と明確に分かれており，そのインターフェイスが最小であるプログラム

このうち，①の項目はC言語の処理系の違いやCPUの違いを吸収するときに必須のものであり，②の項目は機種(ハードウェアやオペレーティングシステム)の違いを吸収するためのものです．

①については1.1節でも解説しましたから，あまり説明の必要はないでしょう．これはプログラムを組むうえで当たり前のことでもあるからです．型を明確に定義した変数とポインタ変数は移植性を高めるばかりでなく，プログラムを組むうえでの大きな間違いを減らし，わけのわからない暴走といった事態を防いでくれます．

以下では②の場合についてくわしく解説します．

■ 移植性の高いプログラムの書き方

次ページの図1-11を見てください．左側の構造図のプログラムでは画面の入出力をたった2つの関数として実現し，この部分のみが機種やオペレーティングシステムに依存した部分になります．これはたまねぎの皮のような構造をしており，その芯は機種，オペレーティングシステムに依存した部分があり，その外側は機種やオペレーティングシステムに依存しない部分になります．つまり，この構造図では画面への出力にはかならず「1文字出力関数」を使わなければならないという〝規則〟のもとにプログラムを作ります．

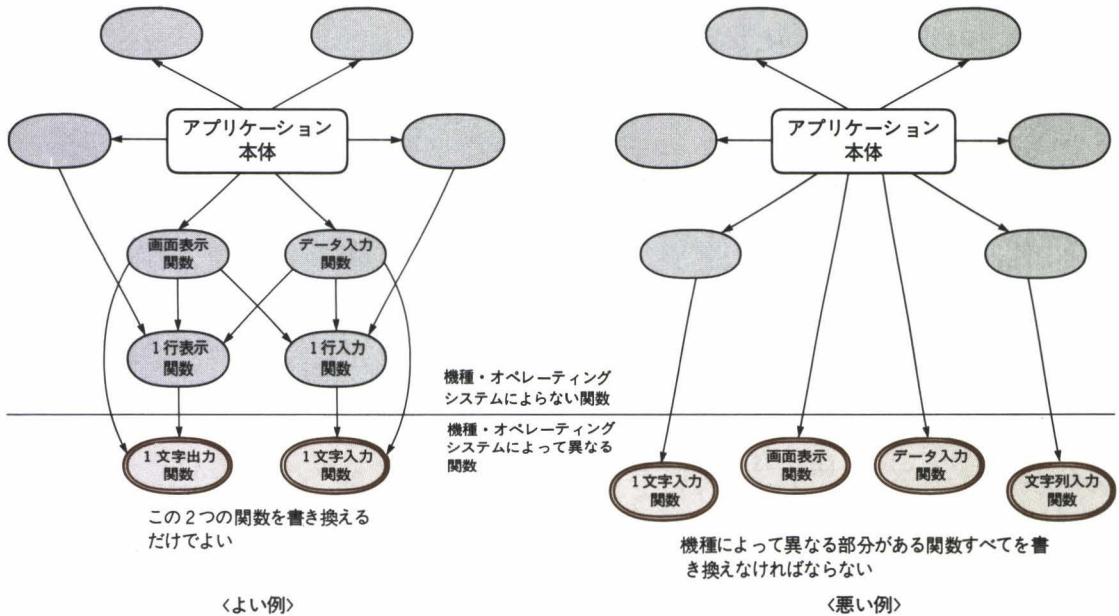



図 1-11　プログラムの構造図

これに対して，右側の構造図のプログラムでは，一見わかりやすそうな構造をしていますが，多くの関数が機種やオペレーティングシステムに依存してしまいます．

左側の構造図で示した約束のもとにプログラムを作れば，プログラムを別の機種やオペレーティングシステムに移植する場合，この2つの，それも単純な機能であるために記述が容易なプログラムを同じ「呼び出し手順」で作り直せばよいだけになります．つまり，この章の1.2節で取り上げた「dspk1()」という関数のみを機種別に書き直せば，どの機種でも画面表示に関しては問題がなくなります．

これらを機種別のライブラリとして整理しておけば，今度はほかのプログラムをこの機種に載せたい場合でも同じライブラリを使えばよいことになります．もちろん，そのためにはドキュメントの整備も必要です．

リスト1-1のdspk1関数を使ったプログラムとして**リスト1-6**のようなプログラムを考えると，このプログラムは機種やオペレーティングシステムが変わってもいっさい手をつけずに利用できます．

```
 1: /*
 2:      Display 1-line procedure
 3: */
 4:
 5: #include     "disp.h"
 6:
 7: int       maxcol     = 80;
 8: int       maxlaw     = 24;
 9:
10: int       cur_color = 7;······················色初期値(白)
11: int       cur_attr  = 0;······················属性初期値(ノーマル)
12: int       cur_law   = 0;······················行初期値(0)
13: int       cur_col   = 0;······················列初期値(0)
14:
15:
16: /*
17:      initial this liblaries·············ライブラリの初期化
18: */
19: void      disp_init(int lmax, int cmax, int color, int attr)
                        └行最大値  └列最大値  └初期色    └初期属性
20: {
21: maxlaw     = lmax;
22: maxcol     = cmax;
23: cur_color = color;
24: cur_attr  = attr;
25: cur_law    = 0;
26: cur_col    = 0;
27: }
28:
29: void      disp_line(int law, int col, unsigned char *line, int color,
    int attr)           └列      └行      └1行の文字データ        └色
        └属性
30: {
31: for(; *line != (unsigned char)0 ; ++line)
32:      {
33:      if(*line >= 0x80)···········シフトJISの1バイト目
34:           {
35:           unsigned int    c;
36:
37:           c  = (*line << 8) & 0xFF00;
38:           line++;
39:           c |= *line & 0x00FF;
40:           dspk1(law,col,c,color,attr);
41:           col += 2;
42:           }
43:      else
44:           {
45:           dspk1(law,col,*line,color,attr);
46:           col++;
47:           }
48:      }
49: }
```

```
50:
51: /*
52:     Write 1-line ············1行表示
53: */
54: void    disp_1(unsigned char *line)
55: {
56: disp_line(cur_law,cur_col,line,cur_color,cur_attr);············行の表示
57:
58: cur_col += strlen(line);············行と列の修正
59:
60: if(cur_col >= maxcol)
61:     {
62:     cur_col -= maxcol;
63:     cur_law += 1;
64:     if(cur_law >= maxlaw)
65:         cur_law = 0;
66:     }
67: }
68:
69: /*
70:     Change Display position
71: */
72: void    disp_pos(int law, int col)
73: {
74: cur_law = law;
75: cur_col = col;
76: }
77:
78: /*
79:     Canege Color
80: */
81: void    disp_color(int color)
82: {
83: cur_color = color;
84: }
85:
86: /*
87:     Change Attributes
88: */
89: void    disp_attr(int att)
90: {
91: cur_attr = att;
92: }
```

リスト 1-6　DISPLAY.C

```
 1: #include <stdio.h>
 2: #include "disp.h"
 3:
 4: main()
 5: {
 6:     int i,law,col,color,attr;
 7:     unsigned char *str = "Test";
 8:
 9:     disp_init( 80, 24, NORMAL );
10:     disp_pos( 5, 5 );
11:     disp_color( C_GREEN );
12:     disp_1( "正常動作" );
13:
14:     for ( law = 10; law <= 17; law++ )
15:         for ( col = 0; col <= 7; col++ ) {
16:             color = law % 7 + 1;
17:             attr = col;
18:             disp_line( law, col * 10, str, color, attr );
19:         }
20: }
```

リスト 1-7　TEST2.C

プログラムの移植は，たいていの場合，時間も手間もかかるものです．しかしプログラムの作り方次第でどうにでもなるものでもあります．とくにC言語の場合は，UNIXという最初から移植を前提に考えられたオペレーティングシステムを記述するために作られたものなので，そのつもりになれば移植を前提としたプログラムは書きやすいはずです．しかし，図1-11の右側の構造図に示したようなプログラムは，たとえC言語で書かれていたとしても，移植が楽とはいえなくなってしまいます．

要するに，移植を前提としたプログラムを作るためには，最初からそのように考えてプログラムを設計する必要があるということです．

1.5　プログラム記述のノウハウ

C 言語はアセンブラのようにきめ細かなプログラムを組むことが可能です．以下にその例としてプログラムの高速化の手法，バグの回避方法，そして最後にシステム設計とプログラム開発ツールについて簡単に説明します．

■ プログラムの高速化

プログラムの実行速度に寄与する最大の要素はなんといってもそのアルゴリズムでしょう．最近のように CPU そのものの速さがミニコン程度のコンピュータでも高級なパーソナルコンピュータ(80386 や 68020 を搭載したもの)でもそう変わらなくなってきた現状では，とくにアルゴリズムの選択が重要になります．そして，その次に速度に関係してくるのはハードウェアの速度，とくに入出力(ディスクとメモリとのやりとりや画面に文字を書く)のスピードが効いてくることが多いようです．

コーディングのよしあしが速度に関係してくるということはあまり期待できないことですが，ここではそれも考えてみましょう．

1. プログラムのアルゴリズムの検討

たとえば，プログラムのなかでファイルを読み込んでくるところがあったとしましょう．この場合，小さなファイルならば，メモリ上にそのファイルの内容をすべて読み込んでからその情報を使うというやり方の方が，いちいちファイルのある部分を読んでからその部分の情報の操作をするよりも高速になります．これは「ディスクドライブが機械である」ということからくる性質です．一定の動作を行うことと，いちいち動作を変えながら動くことでは，当然，一定の動きをまとめてやった方が速いに決まっています．

また，さまざまな操作をできるだけメモリ上で行うという設計も大切です．ただし，この場合は「なるべく少ないメモリを使う」という別のセオリーも立てておかなければならないことも多いので，なかなかうまく両立できないことがあるでしょう．こういった場合は，やはり速度とメモリ容量をにらんだバランスが大切になってきます．

プログラム設計時には「どこが速度を必要とし，どこが必要でないか」という見極めも大切です．エディタのようなプログラムではその初期化に少し時間がかかっても，実際にキーボードを打つときには，なるべく速い速度での実行が望まれます．さらに，リアルタイムにプログラムを動かす必要があるのか，非同期でよいのか，割り込みは使ってよいのかといった要素も見逃せません．

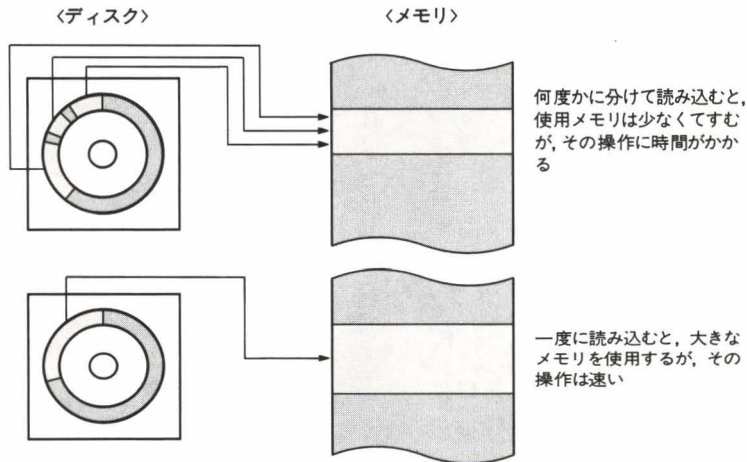



図1-12　ディスクの読み書きの概念

2. ハードウェアの選択

同じオペレーティングシステムが載っているコンピュータでも，ハードウェアによって速度はかなり違います．とくにUNIXなどのハードディスクベースのシステムでは，頻繁にスワップイン，スワップアウトを繰り返すこともあるので，ディスクの速度はCPUの速度よりも問題になる場合があります．

3. プログラムのコーディング上の高速化

高速化の最後はコーディングですが，最近MS-DOS上のコンパイラでは高度なオプティマイザ(最適化プログラム)がついており，人間がちょっと考えつきそうな最適化はかなり行われているようです．たとえば，

```
i = i + 1;
```

というコーディングは以下のようなアセンブラ命令になります．

```
MOV    AX, i
MOV    BX, 1
ADD    AX, BX
MOV    i, AX
```

しかし，ここで足している数は１と決っているわけですから，インクリメント演算子を使って以下のようにコーディングすることもできます．

```
i++;
```

この場合は，次のようなアセンブラ命令に翻訳されます．

```
MOV    AX, i
INC    AX
MOV    i, AX
```

つまり，前者のコーディングの方がステップ数がかかるので，速度も遅いのです．最近はこの程度の最適化であれば，コンパイラがやってくれることが多くなりました．

こういったたぐいの高速化は，たとえばグラフィックのアプリケーションなどで何万回と回るようなループのなかでは，かなり効いてくることもあります．そこでコンパイラが出力したアセンブラプログラムを，さらに人の手で最適化することも行われます．このような場合には「どのようなC言語のコードがどのようなアセンブラに翻訳されるか」といった程度の知識は最低限必要です．

■ バグの回避

プログラムを書いていると，途中で演算子の優先順位などが問題となることがあります．しかし，こういった場合は，優先順位の表などながめずに，迷わずかっこでくくってプログラムを書くことをお勧めします．これはコンパイラ自体のバグの回避になるばかりでなく，計算の順位が明確になるので，デバッグのときにも苦労が少なくてすむからです．

たとえば，次のようなコーディングでは，なにが実行されるのかわからなくなることがあるでしょう．

```
i = *line++;
```

ところが，以下のようなコーディングにするとすっきりします．

```
i = *line;
line++;
```

このような例はC言語に関するかぎり枚挙にいとまがないくらいですが，どんなにへたなコーディングでも動くものであれば，スマートだが動かないプログラムよりもよいのです．

Cコンパイラは，コンパイラとしては比較的単純な構造をしていますから，そのバグも多くないと考えてもよいでしょう．したがって，コンパイラのバグよりもそのコンパイラを使っている人の〝バグ〟の方が圧倒的に多いはずです．とくにコンパイラが解釈に困るようなコーディングは，避けるという

ことを十分考えにいれなければなりません．プログラムにはかならずバグがつきものですから，バグの取りやすいプログラムを書くということに気をつける方が，バグのないプログラムを書くという以上に大切です．

■ システム設計とプログラム開発ツール

C言語独特のシステム設計法というものはありませんが，フローチャートなどの図式化したプログラムの設計という過程はかならず踏んでおくとよいでしょう．最近ではC言語などの構造化言語に適した数々の表現手法(PAD図とかNSチャートなど)が出てきていますから，これらのお世話になるのもよいでしょう．

また，ドキュメントやソースリストの管理にもかなり多くのコンピュータ上のツールが整ってきています．これらツールを使いこなすということは，プログラムの生産性を高めるうえで非常に重要なことです．

表1-4にUNIXシステムでC言語のプログラム開発に使われる主なツールを紹介しましょう．このうちのいくつかは，MS-DOSにも移植されています．

ツール名	機　　能
make	プログラムのコンパイル／リンクの自動化ツール(第7章参照)
SCCS (RCS)	Source Code Control Systemの略．プログラムのソースコードやワープロの文書ファイルの履歴を管理し，いつでも遡った時点のファイルに戻すことができる
grep	複数のファイル中の同一の文字列の検索
sed	ストリーム・エディタ．文字列の置換等をフィルタとして行うことができる
cflow	C言語の関数の呼び出し関係をC言語のソースリストから作成
cb (indent)	きたないC言語のソースにきれいなインデントを付けてくれるツール．どんなプログラムでもきれいにするが，もちろん動くようにしてくれるわけではない
diff	2つ以上のソースリストの差分を出力．差分情報とソースリストからソースリストを再構築してくれるという機能もある
spell	日本のプログラマの苦手な英語の単語のスペルミスをチェック．恥をかく前にぜひ動かしておきたいツール
xref	関数名や変数名のクロスリファレンスリストの生成

表1-4　プログラム開発に使われる主な開発ツール

なお，makeコマンドのくわしい解説は，第7章にありますのでそちらも参照してください．

第2章
オリジナルライブラリの作成

私たちは，C 言語でのプログラミングにあたって，多くの場合，標準ライブラリのお世話になっています．しかし，プログラミングにもある程度慣れてくると，よく使う汎用的な関数をまとめた自分だけのライブラリ（サブルーチン・ライブラリ）を作りたくなるでしょう．

こういったライブラリの作成や管理はライブラリマネージャというプログラムによって行われます．この章では，実用的な汎用ライブラリを作成しながら，ライブラリマネージャの機能と使い方について解説していくことにします．

2.1 ライブラリとは

私たちがC言語でプログラムを記述する際，かならずライブラリ関数のお世話になります．どのライブラリを使うかは，LINK コマンドで指定します．たとえばMS-Cでは次のようなライブラリがあらかじめ付属してきます．

SLIBCE.LIB，GRAPHICS.LIB ……

これらはリンク作業でプログラムに付加されるので，printf 関数のような自分で作成していない関数でも利用できます．英語でのライブラリ(library)には「図書館」という意味がありますが，C言語でも同じように，保存されている多くの関数のなかから必要なものを借りてきて自分のプログラムに組み込んで使えるのです．借りてきたライブラリ関数には，その仕様があらかじめ記述されていますから，それに基づいてプログラムを記述していくことになります．次ページの**図 2-1** には，これらのライブラリの概念を示します．

ライブラリは，実際には複数のリロケータブル・オブジェクトファイル(拡張子に「OBJ」のつくファイル)を1つのファイルに集めたものです．たとえば，「SLIBCE.LIB」というライブラリには，printf 関数や getchar 関数をはじめとする標準関数のリロケータブル・オブジェクトファイルが多数入っているわけです．ライブラリには，このほかにもいろいろな種類があり，先の例でいえば「専門図書館」のように目的別に分類整理されています．

また，ライブラリ中のオブジェクトファイルでプログラムから参照されないものはリンカが自動的に識別しリンクしないようにしてくれますから，プログラムのサイズが不必要に大きくなることはありません．

通常，「C言語でプログラムを組む」といった場合，メーカー供給のライブラリ関数を使うことがほとんどですが，これらのライブラリももとをただせば，私たちがプログラムを組むのとまったく同じ手順でプログラムをコンパイルし，オブジェクトファイルを作り，それを標準ライブラリとしてまとめているものなのです．ライブラリの作成にはとくに難しい点はなく，自分用のライブラリをだれでもすぐに作ることが可能です．リンク時には，こういった標準ライブラリや同じ手順で作成した各自のライブラリを自由に選択できるので，プログラムの目的に応じて各種のライブラリ関数が使えることになります．

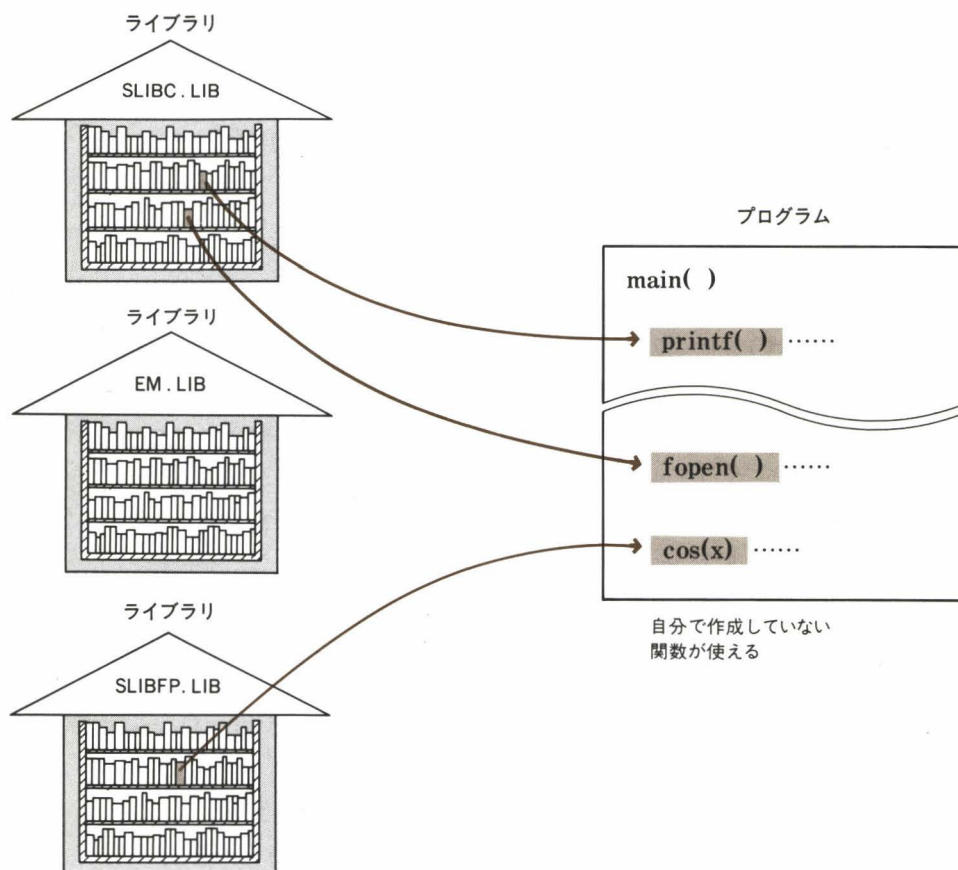



図 2-1　ライブラリの概念

とくにＣ言語でのライブラリは，FORTRAN でのライブラリのようにプログラマ自身の財産となるものであり，ライブラリをいかにうまく効率よく作るかということがプログラム作りの効率を左右します．したがって，どんなＣ言語の処理系であっても動作するプログラムを作ること（移植性のよいプログラムを作ること）は，第１章でも述べたように，プログラムの作成やデバッグが楽になるというだけでなく，ライブラリとして残る自分の財産を増やすということにもつながります．

こういった肝心なライブラリにも，バグがあってはお話になりませんし，うまくモジュール化されたプログラムは，そのモジュール１つ１つがプログラマとしての財産ですから，ライブラリの保守／管理ということにも気を配っておきたいものです．

2.2 ライブラリ・マネージャの基本機能

汎用ライブラリを実際に作成するまえに，ライブラリを管理するコマンドであるライブラリ・マネージャの操作方法について実習しておきましょう．ここで紹介するLIBコマンドは，MS-DOSのシステムディスクや各コンパイラに付属しています．

ライブラリ・マネージャは以下のような機能を持っています．

①複数のリロケータブル・オブジェクトファイルを1つのライブラリファイルにまとめる
②ライブラリファイルから，指定したオブジェクトファイルの削除，追加，置換，抽出を行う
③ライブラリファイル内のpublicシンボルのリストを作成する

ライブラリ・マネージャの機能を図にしてみると，図2-2のようになります．

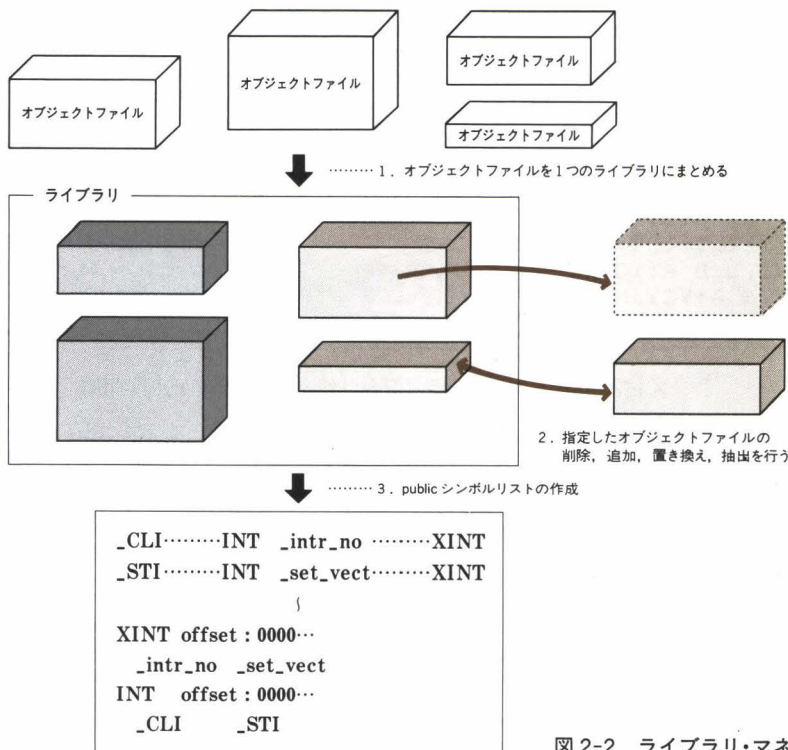



図2-2　ライブラリ・マネージャの機能

これらの操作のうち，①と②はなにをしているか簡単におわかりだと思います．つまり複数のオブジェクトファイルの結合と，そのメンテナンスの実行です．

③は，**マップファイル**の作成です．public シンボルというのは，あるオブジェクトファイルにある関数や変数のうち，そのモジュール以外でも参照できるシンボルの名前とアドレス情報のことです．C 言語では，extern 宣言で参照できる変数や関数を意味します．これを目に見えるテキストファイルの形で，リンクされたあとの情報(すなわち実際のメモリ上での配置を示している)としてまとめたものが，マップと呼ばれるファイルです(リンカが出力する)．このマップによって細かいアセンブラレベルのデバッグのための情報を得るわけです(マップファイルを使ったデバッグについては，第7章を参照)．

マップ情報によって，ライブラリで管理されているリロケータブル・オブジェクトモジュールの一覧が手に入りますから，これを使って多数のオブジェクトによって構成されているライブラリの管理が行えます．

リスト 2-1 にライブラリを作成するバッチファイルを示します．これは MS-C とアセンブラを使って作られたオブジェクトファイルを1つのライブラリにまとめるものです．このライブラリは，2.4 節にある割り込みサポートライブラリです．具体的なプログラムは 2.4 節で解説しますので，ここではライブラリの作り方を理解してください．

```
MASM /MX INT.ASM; ······················INT.ASMをアセンブル
CL -c -AL -Gs -Od -Zd -W1 -G0 XINT.C······XINT.Cをコンパイル
IF EXIST XINT.LIB DEL XINT.LIB··········すでにXINT.LIBがあれば消去
LIB XINT+XINT+INT, XINTLIB.LST··········LIBコマンドを起動し，ライブラリにまとめる
COPY XINT.LIB A:¥C¥LIB¥XINT.LIB·········できたライブラリを必要なディレクトリにコピー
COPY INT.H A:¥C¥INCLUDE¥INT.H···········ヘッダーファイルを必要なディレクトリにコピー
```

リスト 2-1　MKLIB.BAT(ライブラリ作成のバッチファイルの例)

ここで LIB のコマンド行は**図 2-3** のような意味を持ちます．

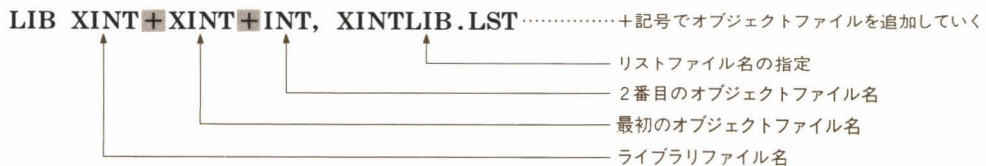



図 2-3　LIB コマンドのオプションの意味

このコマンドの実行後に作成されるリストファイルは，先にも説明したようにこのライブラリの含んでいるオブジェクトファイルに関するすべての情報を出力するファイルです(**図 2-4**)．必要がなければ XINTLIB.LST のかわりに NUL を指定しておきます．

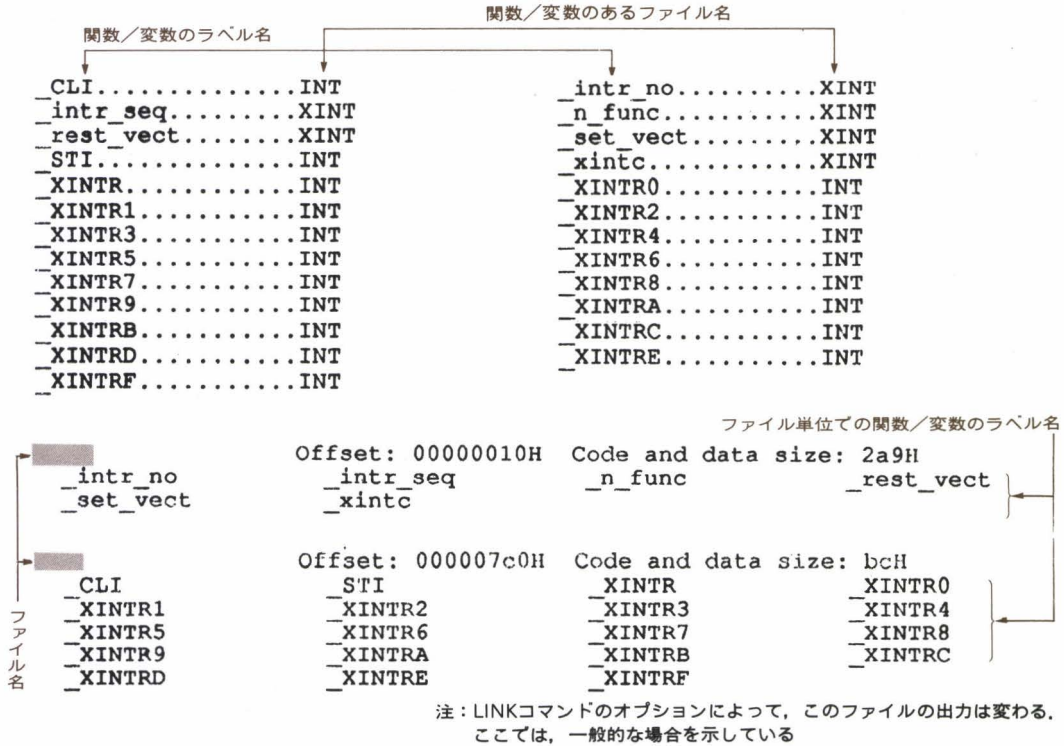



図 2-4　リストファイルの内容

2.3　ライブラリの保守／管理

ここではMS-DOSでのライブラリ管理のうち，LIBコマンドを使ったMicrosoft社フォーマットのライブラリ管理についてまとめておきます．同じMS-DOS上でも，処理系によっては異なった管理方法をとるものもありますから注意してください．

また，UNIXでのライブラリ管理についても簡単にふれておきます．

■ LIBコマンドの使い方

ライブラリの一部をなんらかの理由やバージョンアップによって，交換する必要が出てきた場合を考えてみましょう．リスト2-1で示したバッチファイルを最初から実行してしまうのも悪くはありませんが，多くのオブジェクトファイルが含まれている場合，時間がかかってしまうことでしょう．そこで，1つ(あるいは数個)の比較的少ないライブラリ中のオブジェクトファイルを置き換える場合は，次のようなLIBコマンドを実行するだけで，その操作を行うことができます．

＜前述のXINT.LIB中のINT.OBJを置き換える場合＞

```
A>LIB XINT－＋INT，XINTLIB.LST
```

これはXINT.LIBというライブラリ中のINT.OBJというオブジェクトファイルを置き換える場合のコマンドです．「－＋」というオプションは，「抜いて，加えろ」ということです．つまり，このコマンドは次のように書くこともできます．

```
A>LIB XINT－INT＋INT，XINTLIB.LST
```

これは「XINT.LIB中のINT.OBJを抜いて，外部にあるINT.OBJに入れ換えろ」ということになります．もちろん，単にいらなくなったオブジェクトファイルを抜き取る場合は次のようにするわけです．

```
A>LIB XINT－INT，XINTLIB.LST
```

こういったライブラリは，通常のアプリケーションプログラムの作成時にはそれほど作ることはありませんが，会社単位の共有財産や自分だけのソフトウェア資産として，普遍的なライブラリを作る場合などは必要なものです．

ここで解説したようにLIBコマンド自身の操作は非常に簡単です．LIBコマンドの機能について**表2-1**にまとめておきましょう．

コマンド行での書式

LIB oldlibrary《/PAGESIZE : number》《commands》《,《listfile》《,《newlibrary》》》《 ; 》

oldlibrary　：処理するライブラリファイルの名前
commands：ライブラリファイル上で実行する操作を示すコマンド
/PAGESIZE：ページサイズ（デフォルトでは16バイト）の変更
listfile　：ライブラリ内のモジュールとシンボルのリストファイル
newlibrary：新しいライブラリを作成するときの名前

コマンド文字

コマンド文字	機　　能
+	与えられたライブラリにオブジェクトファイルまたはライブラリファイルを加える
−	ライブラリからモジュールを削除する
−+	モジュールを削除し，同一のオブジェクトファイルを加えてモジュールの置き換えを行う
*	ライブラリからモジュールを抽出し，オブジェクトファイルとして出力する
−*	ライブラリからモジュールを抽出し，オブジェクトファイルとして出力した後，モジュールをライブラリから削除する
;	残りのプロンプトに対して，デフォルトの応答をする
&	現在行を拡張し，コマンドプロンプトを繰り返す

表2-1　LIBコマンドの書式

■ UNIXでのライブラリ管理

UNIXでのライブラリの作成方法について，簡単に紹介しておきます．UNIXのライブラリの管理は，複数のファイルを1つのファイルにまとめるar(アーカイブ)コマンドと，arコマンドで作られたオブジェクトモジュールの集まりにインデックスをつけて参照を容易にするranlibコマンドを使います．次ページの図2-5にライブラリの作成手順を示します．なお最近ではranlibコマンドのないUNIXも増えています．

UNIXに限ったことではありませんが，ライブラリを作った場合は，そのライブラリを利用するためのヘッダーファイルも必要になってきますから，こちらも整備しておく必要があります．また，中身が複雑なものであればあるほど，ドキュメントの存在も欠かせません．ライブラリを作ったらかならずドキュメントの整備もしっかりしておきましょう．

```
% ls -l↵ ………………ライブラリを作るファイルを表示
total 46
-rw-r--r--   1    434 Jan  1 13:54 Makefile
-rw-r--r--   1    855 Jan  1 14:26 jc.h
-rw-r--r--   1    360 Jan  1 13:12 jcdef.h
-rw-r--r--   1   5028 Jan  1 14:36 jclib.c
-rw-r--r--   1   1205 Jan  1 13:12 jctype.h
-rw-r--r--   1   7108 Jan  1 12:52 jslib.c
-rw-r--r--   1    958 Jan  1 13:12 jstring.h
-rw-r--r--   1   4729 jan  1 13:49 knj.c
%
% cat Makefile↵ ………………ライブラリを作るためのMakefileを表示（makeについては第7章を参照）
#
#       Make Japanese-Support Libraries
#       Modified to XENIX/286-sysV
#
#
CFLAGS =        -O -s -Mel ………………コンパイル時のフラグ
OBJS   =        jclib.o jslib.o knj.o ………………ライブラリとするオブジェクト
HEAD   =        jc.h jcdef.h jctype.h jstring.h ………………必要なヘッダーファイル
JLIB   =        Llibj.a ………………作成するライブラリ名

$(JLIB) :       $(OBJS) $(HEAD)
        rm -f $(JLIB)………………古いライブラリを消去
        ar qv $(JLIB) $(OBJS)………………アーカイブ
        ranlib $(JLIB)………………シンボルテーブルを加える
%
% make↵ ………………makeコマンドの実行
        cc -O -s -Mel -c jclib.c
jclib.c
        cc -O -s -Mel -c jslib.c
jslib.c
        cc -O -s -Mel -c knj.c
knj.c
        rm -f Llibj.a
        ar qv Llibj.a jclib.o jslib.o knj.o
ar: creating Llibj.a
q - jclib.o
q - jslib.o
q - knj.o
        ranlib Llibj.a
%
% su↵ ………………ライブラリが作成されたので，rootになり必要なところにコピーする
Password:

% cp ./*.h /usr/include/j/↵ ……………… ヘッダーファイルをコピー
% cp ./*.a /lib/↵ ……………… ライブラリファイルをコピー

% exit↵
%
```

図2-5　UNIXでのライブラリの作成手順

2.4 実用ライブラリの作成

この節では，実用になるライブラリを作ってみましょう．題材としては，MS-DOS上のMS-Cとアセンブラを使って，汎用的に使える「割り込みサポートライブラリ」を作成します．

コンピュータの割り込み処理についてのくわしい説明は第5章で扱っていますので，ここでは省きます．必要があればそちらをお読みください．ここでは，「コンパイル(アセンブル)されたオブジェクトファイルをいかにライブラリとしてアプリケーションプログラムから使うか」ということに主眼を置いて解説していきます．

■ 割り込みサポートライブラリの作成

割り込み処理を記述するプログラムで，必要になる汎用的な関数には，**表2-2**のようなものが考えられます．

関数名	機　　能
set_vect( )	指定したベクタ番号のインタラプトベクタの書き換えと，必要なインタラプトサポートC言語関数のセット
rest_vect( )	set_vect関数でセットしたベクタをもとに戻す

表2-2　割り込み処理で使われる汎用的な関数

このライブラリは一部の関数がアセンブラで記述されており，割り込みをサポートするプログラムをC言語で簡単に記述するためのものです．

UNIXでは，割り込みについてはすべてオペレーティングシステム(OS)でサポートされており，その割り込み要因もOSの管理下にありますから，こういった関数は必要ありません．しかし，MS-DOSなどで割り込みを使ったプログラムを記述する場合には，ここで示すようなインタラプトベクタの書き換えを直接行うプログラムが必要になってきます[*1]．この関数自身はMS-Cとアセンブラで記述されていますが，多少手直しすれば，ほかのコンパイラでもコンパイルすることが可能です(**リスト2-2**，**リスト2-3**のコメント参照)．

この2つのライブラリ関数の動作を図示しておきましょう(**図2-6**)．

＊1　最近ではこの種のライブラリがコンパイラに付属している(Turbo C，MS-C Ver5.0以降など)．

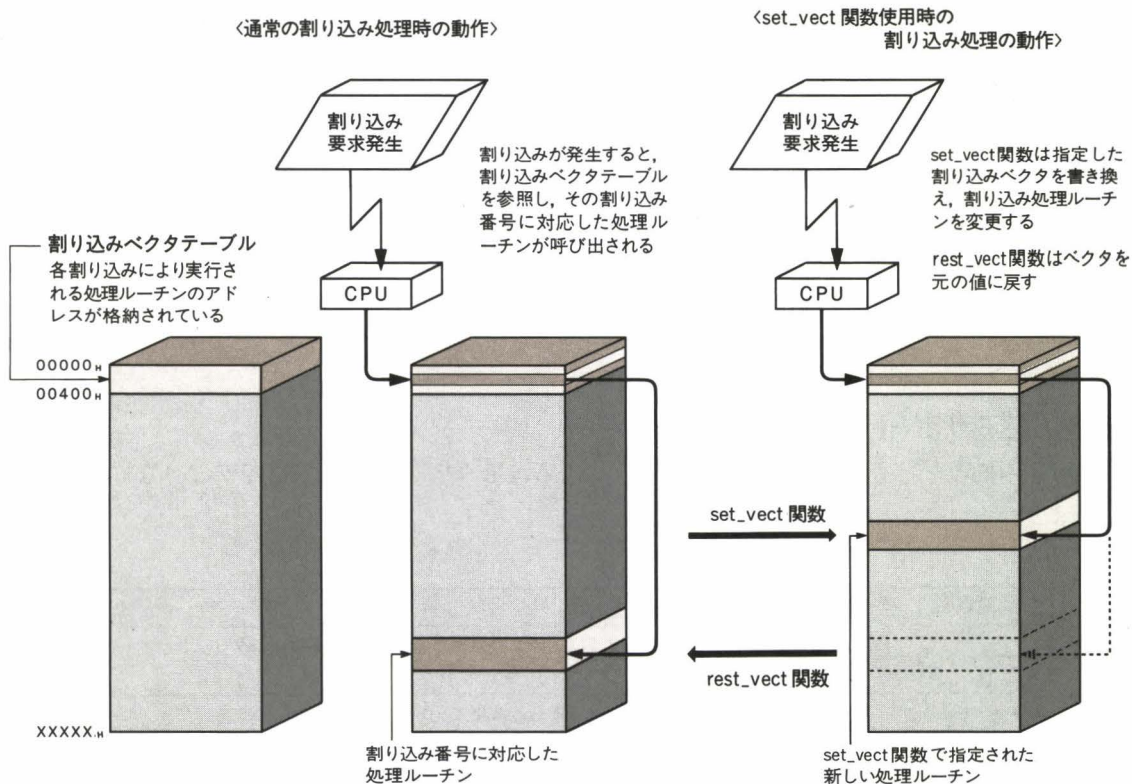



図 2-6　set_vect ／ rest_vect 関数の動作

以下のリスト 2-2 とリスト 2-3 にライブラリのソースファイルを示します．

```
 1: /*
 2:     Interrupt Support Procedure for iAPX86/286 systems
 3: */
 4:
 5: #define     BOOL        int         } 真理値の定義
 6: #define     FALSE       0
 7: #define     TRUE        1
 8:
 9: #define     INT_DOS     (int)0x21…………MS-DOSのシステムコールのインタラプトベクタ
10: #define     GET_VECT    (unsigned char)0x35……ベクタの収得（ファンクション35h）
11: #define     SET_VECT    (unsigned char)0x25……ベクタのセット（ファンクション25h）
```

```
12:
13: #define     MAXINT      16 ……………書き換えられるインタラプトベクタの最大数
14:
15: #include    <stdio.h>
16: #include    <dos.h>
17:
18:
19: struct _vect    { ……………ベクタ値をストアしておく構造体
20:         unsigned char   num;    /* Vector Number */
21:         unsigned short  v_off;  /* Offset Value */
22:         unsigned short  v_seg;  /* Segment Value */
23:         };
24:
25:
26: extern  void    cli(); ……………CLI命令を実行する関数
27: extern  void    sti(); ……………STI命令を実行する関数
28: extern  void    xintr();    /* for Interrupt Support */
29:
30: extern  void    xintr0();
31: extern  void    xintr1();
32: extern  void    xintr2();
33: extern  void    xintr3();
34: extern  void    xintr4();
35: extern  void    xintr5();
36: extern  void    xintr6();
37: extern  void    xintr7();
38: extern  void    xintr8();
39: extern  void    xintr9();
40: extern  void    xintrA();
41: extern  void    xintrB();
42: extern  void    xintrC();
43: extern  void    xintrD();
44: extern  void    xintrE();
45: extern  void    xintrF();
46:
47: void    (*n_func[])() = {   xintr0,xintr1,xintr2,xintr3,
48:                 xintr4,xintr5,xintr6,xintr7,
49:                 xintr8,xintr9,xintrA,xintrB,
50:                 xintrC,xintrD,xintrE,xintrF,
51:                 (void *)NULL
52:             };
53:
54: void    (*m_func[MAXINT])();
55:
56: union   REGS    inp_reg;
57: union   REGS    out_reg;
58: struct  SREGS   seg_reg;    /* Resisters */
59:
60: typedef struct _vect    VECT;   /* Set typedef for vector */
61: VECT    sint[MAXINT];       /* interrupt management table */
62:
63:
64: unsigned short  intr_no = 0;    /* Interrupt Number */
65: unsigned int    intr_seq = 0;   /* Sequential Number */
66:
67:
```

（26〜45行）INT.ASMにある関数

（30〜45行）インタラプト番号に対応する関数のアドレス（実は1つ）

（47〜51行）アセンブラで記述した関数へのコール番地の配列

```
 68:
 69: /*
 70:     Called from INT.ASM:xintr procedures
 71: */
 72:
 73: void    xintc(unsigned short num)        /* Called from INT.ASM */
 74:                                          /* num: Called Numbers */
 75: {
 76: (*m_func[num])();              /* Call specified function */
 77: }
 78:
 79:
 80:
 81: /*
 82:     User function : Set vectors for user-interrupt procedure
 83: */
 84:
 85: BOOL    set_vect(int v_num, void (*i_func)())
 86:     /* v_num:  Vector Number */
 87:     /* i_func: Called C-function */
 88: {
 89: if(MAXINT <= intr_seq) return(FALSE);    /* too many interrupts */
 90:
 91: segread(&seg_reg);              /* Get segment value */
 92: inp_reg.h.ah = GET_VECT;           /* Set AH */
 93: inp_reg.h.al = (unsigned char)v_num;     /* Set AL */
 94: int86x(INT_DOS,&inp_reg,&out_reg,&seg_reg)
 95: segread(&seg_reg);              /* Do Interrupt */
 96:
 97: sint[intr_seq].num   = (unsigned char)v_num;
 98: sint[intr_seq].v_off = out_reg.x.bx;
 99: sint[intr_seq].v_seg = seg_reg.es;
100:
101: /* ============= Vectors store Complete ============== */
102:
103: segread(&seg_reg);
104: inp_reg.h.ah = SET_VECT;
105: inp_reg.h.al = (unsigned char)v_num;
106: inp_reg.x.dx = (unsigned short)((long)(n_func[intr_seq]));
107: seg_reg.ds   = (unsigned short)((long)(n_func[intr_seq]) >> 16);
108: cli();
109: int86x(INT_DOS,&inp_reg,&out_reg,&seg_reg); /* Set New Interrupt
     vectors */
110:
111: /* ============= Set Vectors complete ================ */
112:
113: m_func[intr_seq] = i_func;
114: intr_seq++;
115: sti();
116: return(TRUE);
117: }
118:
119:
```

（90行目注）segread関数はコンパイラによって仕様が異なる場合がある

（94行目注）int86x関数はコンパイラによって仕様が異なる場合がある
なお，intdosx関数を使うことも可能
（第4章参照）

```
120: /*
121:      Restore Interrupt vector value
122: */
123:
124: void    rest_vect()
125: {
126: int i;
127:
128: for(i = 0 ; i < intr_seq ; ++i)
129:      {
130:      segread(&seg_reg);
131:      inp_reg.h.ah = SET_VECT;
132:      inp_reg.h.al = sint[i].num;
133:      inp_reg.x.dx = sint[i].v_off;
134:      seg_reg.ds   = sint[i].v_seg;
135:      cli();
136:      int86x(INT_DOS,&inp_reg,&out_reg,&seg_reg);
137:                  /* Set New Interrupt vectors */
138:      sti();
139:      }
140: intr_seq = 0;             /* Reset sequence values */
141: }
```

リスト 2-2 XINT.C

```
 1: ;
 2: ;        Static Name Aliases
 3: ;
 4:          TITLE    INTERRUPTS
 5: ;
 6: ;
 7: INT_TEXT          SEGMENT  BYTE PUBLIC 'CODE'
 8: INT_TEXT          ENDS
 9: ;
10: XINT_TEXT         SEGMENT  BYTE PUBLIC 'CODE'
11: EXTRN    _xintc:FAR
12: XINT_TEXT         ENDS
13: ;
14: _DATA    SEGMENT  WORD PUBLIC 'DATA'
15: _DATA    ENDS
16: ;
17: CONST    SEGMENT  WORD PUBLIC 'CONST'
18: CONST    ENDS
19: ;
20: _BSS     SEGMENT  WORD PUBLIC 'BSS'
21: _BSS     ENDS
22: ;
23: DGROUP   GROUP    CONST,  _BSS,   _DATA
24:          ASSUME   CS: INT_TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
25: ;
26: EXTRN    _intr_no:WORD                    ; set vector no.
27: ;
28: INT_TEXT       SEGMENT
```

（7～21行）セグメントのセットアップ．コンパイラの仕様によって異なるので注意．この部分は，からの関数をコンパイルしてアセンブラソースをはかせ，それを利用してもよい

```
29: ;
30: PUBLIC          _xintr
31: _xintr          PROC FAR
32: ;
33: PUBLIC          _xintr0 ············0番目のインタラプトベクタの指すエントリポイント
34: _xintr0         LABEL FAR
35:                 cli
36:                 push    ax
37:                 mov     ax,0h
38:                 jmp     _xintrx
39:
40: PUBLIC          _xintr1
41: _xintr1         LABEL FAR
42:                 cli
43:                 push    ax
44:                 mov     ax,1h
45:                 jmp     _xintrx
46:
47: PUBLIC          _xintr2
48: _xintr2         LABEL FAR
49:                 cli
50:                 push    ax
51:                 mov     ax,2h
52:                 jmp     _xintrx
53:
54: PUBLIC          _xintr3
55: _xintr3         LABEL FAR
56:                 cli
57:                 push    ax
58:                 mov     ax,3h
59:                 jmp     _xintrx
60:
61: PUBLIC          _xintr4
62: _xintr4         LABEL FAR
63:                 cli
64:                 push    ax
65:                 mov     ax,4h
66:                 jmp     _xintrx
67:
68: PUBLIC          _xintr5
69: _xintr5         LABEL FAR
70:                 cli
71:                 push    ax
72:                 mov     ax,5h
73:                 jmp     _xintrx
74:
75: PUBLIC          _xintr6
76: _xintr6         LABEL FAR
77:                 cli
78:                 push    ax
79:                 mov     ax,6h
80:                 jmp     _xintrx
81:
```

```
 82: PUBLIC          _xintr7
 83: _xintr7         LABEL FAR
 84:                 cli
 85:                 push    ax
 86:                 mov     ax,7h
 87:                 jmp     _xintrx
 88:
 89: PUBLIC          _xintr8
 90: _xintr8         LABEL FAR
 91:                 cli
 92:                 push    ax
 93:                 mov     ax,8h
 94:                 jmp     _xintrx
 95:
 96: PUBLIC          _xintr9
 97: _xintr9         LABEL FAR
 98:                 cli
 99:                 push    ax
100:                 mov     ax,9h
101:                 jmp     _xintrx
102:
103: PUBLIC          _xintrA
104: _xintrA         LABEL FAR
105:                 cli
106:                 push    ax
107:                 mov     ax,0Ah
108:                 jmp     _xintrx
109:
110: PUBLIC          _xintrB
111: _xintrB         LABEL FAR
112:                 cli
113:                 push    ax
114:                 mov     ax,0Bh
115:                 jmp     _xintrx
116:
117: PUBLIC          _xintrC
118: _xintrC         LABEL FAR
119:                 cli
120:                 push    ax
121:                 mov     ax,0Ch
122:                 jmp     _xintrx
123:
124: PUBLIC          _xintrD
125: _xintrD         LABEL FAR
126:                 cli
127:                 push    ax
128:                 mov     ax,0Dh
129:                 jmp     _xintrx
130:
131: PUBLIC          _xintrE
132: _xintrE         LABEL FAR
133:                 cli
134:                 push    ax
135:                 mov     ax,0Eh
136:                 jmp     _xintrx
137:
```

```
138: PUBLIC              _xintrF
139: _xintrF             LABEL FAR
140:                     cli
141:                     push    ax
142:                     mov     ax,0Fh
143:
144: _xintrx:
145:                     push    ds ……………………AXレジスタにインタラプト番号が入ってくる
146:                     push    ax
147:                     mov     ax,DGROUP ┐
148:                     mov     ds,ax     ┘DSレジスタをセットアップする
149:                     pop     ax
150:                     mov     _intr_no,ax      ; Store Jump-Number
151:                     pop     ds
152:                     pop     ax
153: ;
154:                     pushf
155:                     push    ax
156:                     push    bx
157:                     push    cx
158:                     push    dx
159:                     push    bp               ; Save all resisters  } レジスタの退避
160:                     mov     bp,sp
161:                     push    si
162:                     push    di
163:                     push    ds
164:                     push    es
165: ;
166:                     mov     ax,DGROUP
167:                     mov     ds,ax            ; Set DS: to DATA-SEGMENTS
168:                     mov     es,ax
169: ;
170:                     cld                      ; Clear Direction Flag
171: ;
172:                     mov     ax,_intr_no
173:                     push    ax               ; Push interrupt number
174:                     call    _xintc           ; Call C-language Routine
175:                     add     sp,2             ; Return from C-lang
176: ;
177:                     pop     es
178:                     pop     ds
179:                     pop     di
180:                     pop     si
181:                     mov     sp,bp
182:                     pop     bp               ; Restore all resisters
183:                     pop     dx
184:                     pop     cx
185:                     pop     bx
186:                     pop     ax
187:                     popf
188:                     sti
189:                     iret                     ; Return to interrupted
     procedure
190: ;
191: _xintr              ENDP
192: ;
```

```
193: ;
194: ;                    INTERRUPT CLEAR FUNCTIONS
195: ;
196: PUBLIC               _cli
197: _cli                 PROC FAR
198: ;
199:                      cli
200:                      ret
201: ;
202: _cli                 ENDP
203: ;
204: ;
205: ;
206: ;                    INTERRUPT SET FUNCTIONS
207: ;
208: PUBLIC               _sti
209: _sti                 PROC FAR
210: ;
211:                      sti
212:                      ret
213: ;
214: _sti                 ENDP
215: ;
216: ;
217: INT_TEXT             ENDS
218: ;
219: ;
220: END
221: ;
```

リスト2-3　INT.ASM

このライブラリは，XINT.CとINT.ASMという2つのプログラムから作られています．これらのアセンブラとC言語のリンクの問題については，第5章でくわしく解説しますので，ここでは取り上げないことにします．

これらの関数を使うヘッダーファイルは，「INT.H」という名前になっています．**リスト2-4**にINT.Hのリストを示します．

```
 1: /*
 2:         Interrupt support C-Language Routines Header
 3: */
 4:
 5: #ifdef  LINT_ARGS
 6: int     set_vect(int,(*)());
 7: void    rest_vect(void);
 8: #else   /* LINT_ARGS */
 9: extern  int     set_vect();
10: extern  void    rest_vect();
11: #endif  /* LINT_ARGS */
```

```
12:
13: /*
14:
15: 使用法:
16:
17:
18: ●set_vect()関数
19:
20:         BOOL    set_vect(vnum,function);
21:
22:                 int     vnum;           使用するベクタ番号
23:                 void    (*func)()       割り込みがかかったときに
24:                                         動く関数へのポインタ
25:
26:         これらの関数はラージモデルでのみ使うことができます.
27:         この関数は, 指定したベクタ番号からの割り込みがあった場合,
28:         指定したC言語の関数にその制御を移します.
29:
30:         この関数では割り込み要因についてはそれぞれのシステムによって
31:         違うので, いっさい関知していません.
32:         戻り値は割り込みベクタのセットが成功したら, TRUE, 失敗したら
33:         FALSEを返します. また, この関数の使用できる割り込みベクタ
34:         の数は, 16までです.
35:
36:
37:         void    rest_vec();
38:
39:
40:         この関数はset_vect()関数でセットされた新たなベクタを,
41:         すべてset_vect()起動以前の状態に戻します.
42:
43:         戻り値はありません.
44:
45: */
46:
```

リスト2-4　INT.H

各プログラムをコンパイル(アセンブル)し，ライブラリを作成し，さらにできたライブラリとヘッダーファイルを必要なディレクトリにコピーし，いつでも必要なときに使える形に整備しておく必要があります．こういった処理はバッチファイルを使っても行えますが，ここでは，MS-DOSのMASMなどに付属のMAKEコマンドを使ってこの操作を行ってみます(MAKEについてのくわしい説明は第7章を参照のこと)．

次ページの**リスト2-5**で示すMAKEFILEを以下のように起動すると，新しいライブラリとヘッダーファイルが必要なディレクトリに作られます．

```
A>MAKE MAKEFILE
```

このライブラリの作成過程は，次ページの**図2-7**のようになります．

```
INT.OBJ :    INT.ASM
    MASM /MX INT.ASM;

XINT.OBJ :   XINT.C
    CL -AL -Gs -Od -Zd -W1 -G0 -c XINT.C

XINT.LIB :   XINT.OBJ INT.ASM
    COMMAND /C IF EXIST XINT.LIB DEL XINT.LIB
    LIB XINT+XINT+INT,XINTLIB.LST

¥C¥LIB¥XINT.LIB : .¥XINT.LIB
    COMMAND /C COPY .¥XINT.LIB A:¥C¥LIB¥XINT.LIB

¥C¥INCLUDE¥INT.H : .¥INT.H
    COMMAND /C COPY .¥INT.H A:¥C¥INCLUDE¥INT.H
```

-Gs：スタックチェックを省くオプション（割り込み処理では必ず指定）
-Od：オプティマイザを通さない（安全のため）
-AL：ラージモデルでコンパイルすること
-G0：8086のコードを出力する
LIB XINT+XINT+INT,XINTLIB.LST：ライブラリの作成
¥C¥LIB¥XINT.LIB ～ ¥C¥INCLUDE¥INT.H：ライブラリとヘッダーファイルを必要なディレクトリにコピー

リスト 2-5　MAKEFILE

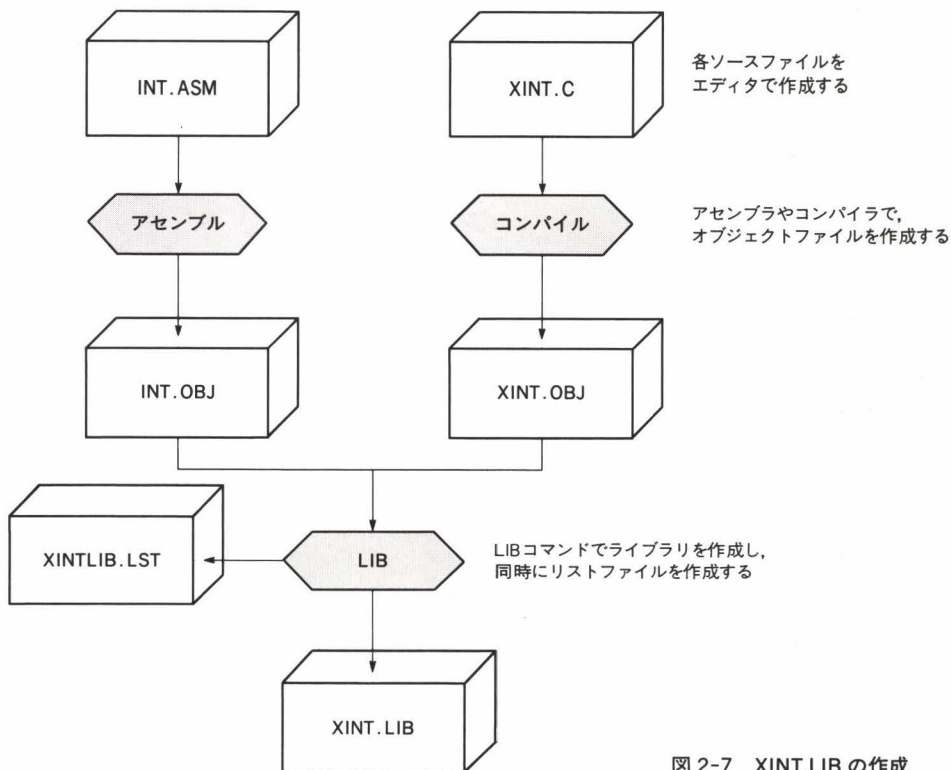



図 2-7　XINT.LIB の作成

こういった方法でできたライブラリは，もちろん，デバッグも必要なことがあります．通常はライブラリとするまえに，オブジェクトファイルの段階でデバッグをし，最後にライブラリにまとめます．

■ 割り込みサポートライブラリの利用

割り込みサポートライブラリを使うには，どうしたらよいのでしょうか？　以下に，これらのライブラリを使った簡単なテストプログラムを組み，実際に使ってみましょう（**リスト2-6**）．

```
 1: /*
 2:      Interrupt Support procedure TEST-functions
 3:
 4:      For PC-9801 Series Personal Computer ONLY
 5: */
 6:
 7: #include     <stdio.h>
 8: #include     <conio.h>
 9:
10: #include     "int.h"       /* Headers for interrupts */
11:
12: int stp,cpy;
13:
14: /*
15:      Interrupt Procedure
16: */
17: void     stpx()
18: {
19: stp++;
20: }
21:
22: void     cpyx()
23: {
24: cpy++;
25: }
26:
27: /*
28:      M a i n
29: */
30: main()
31: {
32:
33: stp = 0;
34: cpy = 0;            /* Clear flags */
35:
36: if(!set_vect(0x6,stpx))
37:      {
38:      printf("¥7**** Cannot Set STOP ****¥n");
39:      exit(0);
40:      }
41:
42: if(!set_vect(0x5,cpyx))
```

STOPキーが押されると，この関数へとぶ

COPYキーが押されると，この関数へとぶ

STOPキーのベクタ

STOPキーの割り込みを使う

他機種に移植する場合は，テクニカルマニュアルを見て書き換える

COPYキーのベクタ

```
43:     {                                          COPYキーの割り込みを使う
44:     printf("¥7**** Cannot Set COPY ****¥n");
45:     exit(0);
46:     }
47:
48: for(;;)
49:     {
50:     if(stp) ………STOPキーが押されたかどうか調べる
51:         {
52:         stp--;
53:         printf("PUSHED [STOP]¥n");
54:         }
55:     if(cpy)………COPYキーが押されたかどうか調べる
56:         {
57:         cpy--;
58:         printf("PUSHED [COPY]¥n");
59:         }
60:
61:     if(kbhit())
62:         {                    他のキーが押されたかどうかを調べ,
63:         getch();             押されたらループを抜ける
64:         break;
65:         }
66:     printf(".");
67:     }
68: rest_vect(); …………ベクタをもとに戻す
69: }
```

リスト 2-6　TESTINT.C

このテストプログラムは，PC-9801専用のもので，STOPキーとCOPYキーの割り込みをアプリケーションプログラムから使えるようにしたものです．短いプログラムなので，くわしくはプログラムリストを見てください．ほかの機種へも簡単に移植できますから，プログラム中のコメントを参照してください．

このTESTINT.Cから，TESTINT.EXEを作るバッチファイルを**リスト 2-7**に示します．

```
CL -AL -Gs -Zd -G0 -W1 -FeTESTINT TESTINT.C XINT.LIB

                                    XINTライブラリをリンクする
```

リスト 2-7　MKTEST.BAT

ここで注目すべきことは，作ったプログラムのリンクの方法です．2行目のLINKコマンドを見ればわかるように，通常の標準ライブラリをリンクするのと，まったく同じ要領でよいのです．

LIB コマンドでのライブラリの作成について見てきましたが，これらのライブラリ操作はプログラム開発者の財産管理をしているようなものですから，ぜひ覚えて使いこなすことをお勧めします．また，こういったユーティリティがあるために，プログラムの組み方もかなり変わってくることもあります．今まではプログラム１つ１つを組んできた方も，こういったあとあとのことを考えた「ダイナミックな」プログラミングが効率的なプログラムを組むことを助けるのだということがわかってくるでしょう．

■ ライブラリ作成の注意点

ライブラリの作り方は理解しても，どのような関数をライブラリにしたらよいのかという問題が残ります．ここではその指針と注意点を整理しておきましょう．

ライブラリを作る場合，ライブラリとする関数は以下のようなものである必要があります．

・誰が使っても有効な内容であること
・グローバル変数をなるべく使わないこと
・関数は，関数名を見ただけでなにをするものなのかすぐに想像がつくものであること
・関数のドキュメントを整備しておくこと．あるいは，豊富なコメントをつけること

内容が特定のアプリケーションプログラムにのみに有効なものであれば，あらためてライブラリ関数としてまとめておく意味はありません．また，グローバル変数を使ったライブラリは使いにくく，アプリケーションプログラムとそのグローバル変数名が重なってしまったときなどは，目も当てられません．また，モジュールごとの独立性もきちんと保てないので，あとで使う場面がなくなってしまいます．

関数の名前も，わかりやすいものがよいと筆者は考えていますが，なかには「func0001()」などと，通し番号で管理する人もいるようです．こういった関数名のつけ方は，関数名や変数名の重複が起きやすい場合などは，有効な手段でもあります．

作った関数のドキュメントはあるにこしたことはありません．ソースファイルに多くのコメントがついていると，あとで見つかるかもしれないバグにはとても有効です．最低限，外部仕様ぐらいは，なんらかの形で残しておく必要があります．

2.5 ヘッダーファイルによる簡易ライブラリの作成

こういった"真面目な"ライブラリばかりではなく，C言語の機能として備わっている「マクロ機能」を使った「マクロ・ライブラリ」というライブラリもあります．これは，マクロの簡単な置き換え機能を用いたものですが，場合によってはかなり有効なライブラリになります．たとえば，エスケープシーケンスを用いた画面のカーソルの制御などは，このようなマクロによるヘッダーファイルを使うのが効果的です．

■ エスケープシーケンスとは

コンピュータのターミナルには，エスケープシーケンスと呼ばれる画面コントロールのための特殊な文字データからなる文字セットが用意されています．この機能はMS-DOSでもサポートされており，画面のクリアやカーソルの移動などがエスケープシーケンスを送ることで実現できます．

このコード群はすべてESC(0x1B)の文字コードを先頭とする数文字分のコントロールコードからなっています．たとえば，「画面を消去し，カーソルを画面の左上の位置に置く」ためには，「0x1B，'[','2','J'」という4文字分のデータを送ります．また，カーソルをある位置にもっていくエスケープシーケンスなどのように，固定した文字列ではなく可変の文字列を扱うものもあります．

これらのエスケープシーケンスは，プログラムを作る側からいうと，以下のような種類に分類できます．

- 固定長エスケープシーケンスでESC以下が1文字のもの
- 固定長エスケープシーケンスでESC以下が2文字のもの
- 固定長エスケープシーケンスでESC以下が3文字のもの
- 可変文字長のエスケープシーケンスのもの

エスケープシーケンスは，DEC社の専用ターミナルであるVT-100のものが多く使われています．実際にはVT-100のエスケープシーケンスは100種類以上ありますが，通常使われているのはこのうちの10種類程度です(**表 2-3** を参照)．第4章で紹介するターミナルエミュレータでもVT-100のエスケープシーケンスをいくつかサポートしています．

フォーマット	デフォルト	解　　　説
ESC [pl ; pcH	1 ; 1	カーソルを，pl行目のpc桁目に移動する
ESC [pl ; pcf	1 ; 1	カーソルを，pl行目のpc桁目に移動する
ESC [pnA	1	カーソルを，pn行上へ移動する
ESC [pnB	1	カーソルを，pn行下へ移動する
ESC [pnC	1	カーソルを，pn桁右へ移動する．行の右端より先には移動しない(次の行へは移動しない)
ESC [pnD	1	カーソルを，pn桁左へ移動する．行の左端より先には移動しない(前の行へは移動しない)
ESC [6n	省略不可	カーソル位置を，直後のコンソール入力(キーボードからの入力)で知らせる．コンソールからの入力形式は ESC [pl ; pcR
ESC [s	引数なし	その時点でのカーソルの位置と文字の表示属性(ESC [mの項を参照のこと)を作業領域に退避する
ESC [u	引数なし	直前に実行したESC [sで退避したカーソル位置と表示属性を呼び出し，セットする
ESC [2J	省略不可	画面をクリアし，カーソルをホームに移動する
ESC [K	引数なし	カーソル位置から，行の右端までを消去する．カーソル位置はそのまま
ESC [ps ; …… ; psm	0	psで指定した表示属性をセットする．これ以降に表示される文字はこの属性にしたがって表示される．psは任意個数指定できるが，機種によっては同時に指定しても無効なものがあるので注意

ps	属　　性
0	メーカーの規定した初期値
1	強調(ハイライト，もしくは太字)
4	下線付き
5	ブリンク(点滅)
7	リバース(反転)
8	シークレット(文字を見せない)
30	黒
31	赤
32	緑
33	黄色
34	青
35	マゼンダ
36	水色
37	白
40	背景　黒
41	背景　赤
42	背景　緑
43	背景　黄色
44	背景　青
45	背景　マゼンダ
46	背景　水色
47	背景　白

表2-3　VT-100エスケープシーケンス(主なものを抜粋)

■ エスケープシーケンス簡易ライブラリ

エスケープシーケンスを使って画面をコントロールする場合は，**リスト2-8**のようにヘッダーファイルにマクロ定義した簡易ライブラリを作っておきます．

```
 1: /*
 2:      Escape Sequence Library Header
 3:      for PC-9801 with MS-DOS
 4: */
 5:
 6: #include      <stdio.h>
 7:
 8: #define       crt_flush()       fflush(stdout)
 9: #define       crt_out(c)        fputs(c,stdout);crt_flush()
10: #define       crt_pos(y,x)      printf("¥033[%02d;%02dH",(y)+1,(x)+1)
11: #define       crt_clear()       crt_out("¥033[2J")
```

リスト2-8　ESC.H

これを使ったカーソルコントロールのサンプルプログラムを以下に示します(**リスト2-9**)．

```
 1: /*
 2:      Escape Sequence Header TEST-Procedure
 3: */
 4:
 5: #include      "esc.h"
 6:
 7: #ifdef        NULL
 8:
 9: #else
10: #include      <stdio.h>
11: #endif
12:
13: void    wait()
14: {
15: long    i;
16:
17: for(i = 0 ; i < 100000 ; ++i);
18: }
19:
20:
21: main()
22: {
23: wait();
24: crt_out("¥nClear Screen.¥n");
25: wait();
```

```
26:
27: crt_clear();
28:
29: crt_out("¥nCoursol Control¥n");
30:
31: crt_pos(0,0);
32: crt_out("Left Up");
33: wait();
34:
35: crt_pos(24,0);
36: crt_out("Left Down");
37: wait();
38:
39: crt_pos(0,70);
40: crt_out("Right Up");
41: wait();
42:
43: crt_pos(24,70);
44: crt_out("Right Down");
45: wait();
46:
47: crt_pos(12,35);
48: crt_out("Center");
49: wait();
50:
51: }
```

リスト2-9　ESCTEST.C

こうして作ったヘッダーによるマクロ・ライブラリは，もちろん単なる文字列の置き換えをするだけなので，その名前の関数名のアドレスはマップファイルを出力させても，いっさい出てきません．ということは，そのマクロの中身をデバッグする必要が生じたとき，デバッガは使えないということですから注意してください．

第3章
日本語処理

現在，コンピュータで日本語を扱えることは当然のこととなっており，国内で販売されるコンピュータで漢字を使用できないものはほとんど見かけなくなってきました．とくにパソコンの場合，〝電子計算機〟ではなく，〝文書処理機〟として使用される割合が年々高まってきていますから，日本では日本語の扱えないパソコンとそのソフトウェアはまともな商品としての市民権が得られないといっても過言ではないでしょう．

しかし，コンピュータの基本ソフトウェアであるOSやプログラミング言語は，いまだに欧米で生れ育ったものがほとんどです．これらの処理系やOSの多くは，アルファベットのみを扱うことを前提に作られたものが多く，日本語を使用するとなるといくつかの工夫が必要になります．

本章では，エラーメッセージを日本語で表示するといった程度のプログラムから，エディタなど日本語の編集を目的としたプログラムにいたるまで，C言語で日本語を使用する際に生じる問題点とそれらの解決方法を示していきます．

3.1　漢字コード体系

日本で使われている漢字は，新聞や公用文書などに使用される常用漢字だけでも千数百種あり，常用漢字以外の固有名詞などに使われているものも含めると，その数は数千のオーダーにまで達します．実際，後述するJISコードでは，使用頻度が高い文字を集めた第1水準で2965種，固有名詞などに使われる第2水準漢字を含めると6353種類の文字が用意されています．

ここでは，JIS規格に定められているJISコードとそれから派生したいくつかの漢字コード体系を，その構成といった点から紹介します．

■ 7ビット系漢字コード

C言語が生まれた米国では，キャラクタコードの標準としてASCIIコードやEBCDICコードなどが用いられています．これらのコードは7ビットで1文字を表していますから，最大で128種類の文字を表現することができます．

MS-DOSマシンが使用しているASCIIコードや，ASCIIコードを手本にして作られた「JIS X 0201情報交換用符号」と呼ばれるJIS規格では，128のコードのうち0x00から0x1Fまでの32個のコードを機能コードとして改行文字などの特殊な文字に割り当てています．そして，スペース(0x20)と削除(0x7F)を除く，0x21から0x7Eまでの94個のコードを英大文字／小文字および「＄％（）［］」などの記号に割り当てています(次ページの**表3-1-a**を参照)．

表3-1-aでわかるように，7ビットで表せる文字はすべて割り当てられてしまっていますから，漢字をそのなかに割り込ませることができません．また，コード体系を8ビットに拡張したところで128文字しか追加することができず，せいぜいカナ文字を割り当てることぐらいしかできません．実際，「JIS X 0201」の8ビット版は，7ビット版のコードにカタカナと句読点などの記号が追加されているだけです(次ページの**表3-1-b**を参照)．

そのうえ，コードを8ビット以上に拡張した場合，拡張したコードを米国流の7ビットコードにしか対応していないプログラム(たとえば8ビット目以上をマスクしているようなプログラム)に与えると，情報が欠落して文字が正常に読み書きできなくなってしまいます．

そこで，考えられた手法が**シフトコード**による文字セット切り替えです．**図3-1**に示すように，シフトコードを使用することで同一のコードに複数の意味を持たせることができるようになります．

上位３ビット→

下位４ビット↓

行＼列	0	1	2	3	4	5	6	7
0	NUL	TC_7(DLE)	(SP)	0	@	P	`	p
1	TC_1(SOH)	DC_1	!	1	A	Q	a	q
2	TC_2(STX)	DC_2	″	2	B	R	b	r
3	TC_3(ETX)	DC_3	#	3	C	S	c	s
4	TC_4(EOT)	DC_4	$	4	D	T	d	t
5	TC_5(ENQ)	TC_8(NAK)	%	5	E	U	e	u
6	TC_6(ACK)	TC_9(SYN)	&	6	F	V	f	v
7	BEL	TC_{10}(ETB)	'	7	G	W	g	w
8	FE_0(BS)	CAN	(	8	H	X	h	x
9	FE_1(HT)	EM	)	9	I	Y	i	y
10	FE_2(LF)	SUB	*	:	J	Z	j	z
11	FE_3(VT)	ESC	+	;	K	[	k	{
12	FE_4(FF)	IS_4(FS)	,	<	L	¥	l	|
13	FE_5(CR)	IS_3(GS)	—	=	M	]	m	}
14	SO	IS_2(RS)	.	>	N	^	n	‾
15	SI	IS_1(US)	/	?	O	_	o	DEL

行＼列	0	1	2	3	4	5	6	7
0	NUL	TC_7(DLE)	(SP)	ー	タ	ミ	↑	↑
1	TC_1(SOH)	DC_1	。	ア	チ	ム		
2	TC_2(STX)	DC_2	「	イ	ツ	メ		
3	TC_3(ETX)	DC_3	」	ウ	テ	モ		
4	TC_4(EOT)	DC_4	、	エ	ト	ヤ		
5	TC_5(ENQ)	TC_8(NAK)	・	オ	ナ	ユ		
6	TC_6(ACK)	TC_9(SYN)	ヲ	カ	ニ	ヨ	未	未
7	BEL	TC_{10}(ETB)	ァ	キ	ヌ	ラ	定	定
8	FE_0(BS)	CAN	ィ	ク	ネ	リ		
9	FE_1(HT)	EM	ゥ	ケ	ノ	ル	義	義
10	FE_2(LF)	SUB	ェ	コ	ハ	レ		
11	FE_3(VT)	ESC	ォ	サ	ヒ	ロ		
12	FE_4(FF)	IS_4(FS)	ャ	シ	フ	ワ		
13	FE_5(CR)	IS_3(GS)	ュ	ス	ヘ	ン		
14	SO	IS_2(RS)	ョ	セ	ホ	゛		↓
15	SI	IS_1(US)	ッ	ソ	マ	゜	↓	DEL

表 3-1-a　半角文字コード表 ＜7 ビット版＞（JIS X 0201 より引用）

上位４ビット→

下位４ビット↓

行＼列	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	NUL	TC_7(DLE)	(SP)	0	@	P	`	p	↑	↑	未定義	ー	タ	ミ	↑	↑
1	TC_1(SOH)	DC_1	!	1	A	Q	a	q			。	ア	チ	ム		
2	TC_2(STX)	DC_2	″	2	B	R	b	r			「	イ	ツ	メ		
3	TC_3(ETX)	DC_3	#	3	C	S	c	s			」	ウ	テ	モ		
4	TC_4(EOT)	DC_4	$	4	D	T	d	t			、	エ	ト	ヤ		
5	TC_5(ENQ)	TC_8(NAK)	%	5	E	U	e	u			・	オ	ナ	ユ		
6	TC_6(ACK)	TC_9(SYN)	&	6	F	V	f	v	未	未	ヲ	カ	ニ	ヨ	未	未
7	BEL	TC_{10}(ETB)	'	7	G	W	g	w	定	定	ァ	キ	ヌ	ラ	定	定
8	FE_0(BS)	CAN	(	8	H	X	h	x			ィ	ク	ネ	リ		
9	FE_1(HT)	EM	)	9	I	Y	i	y	義	義	ゥ	ケ	ノ	ル	義	義
10	FE_2(LF)	SUB	*	:	J	Z	j	z			ェ	コ	ハ	レ		
11	FE_3(VT)	ESC	+	;	K	[	k	{			ォ	サ	ヒ	ロ		
12	FE_4(FF)	IS_4(FS)	,	<	L	¥	l	|			ャ	シ	フ	ワ		
13	FE_5(CR)	IS_3(GS)	—	=	M	]	m	}			ュ	ス	ヘ	ン		
14	SO	IS_2(RS)	.	>	N	^	n	‾			ョ	セ	ホ	゛		↓
15	SI	IS_1(US)	/	?	O	_	o	DEL	↓	↓	ッ	ソ	マ	゜	↓	未定義

表 3-1-b　半角文字コード表 ＜8 ビット版＞（JIS X 0201 より引用）

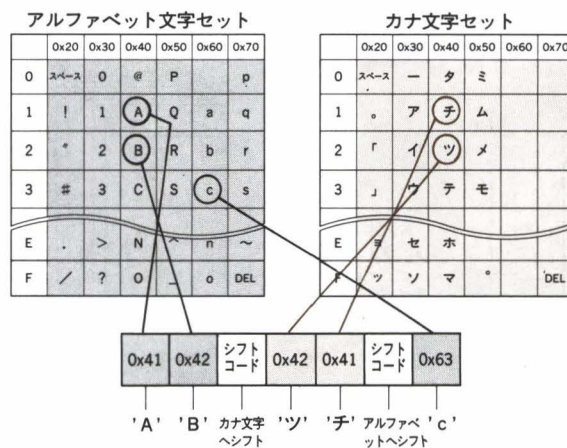



図 3-1　シフトコードによる文字セットの切り替え

図 3-1 の例では 7 ビットコード 1 つがカナ 1 文字に対応する状態を示しましたが，7 ビットコード 2 つで 1 文字を表す状態に移るシフトコードを用意すれば，ASCII の機能コードと重なる部分を避けたとしても，94×94(0x21 から 0x7E までの文字種の 2 乗)で 8836 種の文字を表現でき，漢字も十分表現できることになります(次ページの**図 3-2** を参照)．

シフトコードは日本国内だけでなく，欧米のコンピュータ上でも使用される可能性があります．また，日本語以外にも中国漢字や韓国のハングル文字，アラビア語などのアルファベット以外の文字を扱う場合，なんらかのシフトコードを使って文字列を表現することになります．

これらのシフトコードを，それぞれが勝手に決めてしまうと，ほかの国のデータの読み書きができなくなってしまいますから，ISO(International Organization for Standardization)という国際機構の決めた規格によってシフトコードの割り当てが行われています．日本語の場合，シフトコードを「JIS X 0202」という JIS 規格に定めることで，国内的な裏付けを与えています．

シフトコードによって切り替えられたコード体系中の各漢字ごとのコードの割りつけも規格化されており，「JIS X 0208」に文字とコードの対応が定められています．この JIS コードのなかには，漢字以外にもアルファベットやロシア文字などの漢字以外の文字も含まれており，この 2 バイト文字コードの全体を**全角文字**ともいいます．「JIS X 0208」では漢字を出現頻度に応じて，第 1 水準漢字と第 2 水準漢字に分けています．

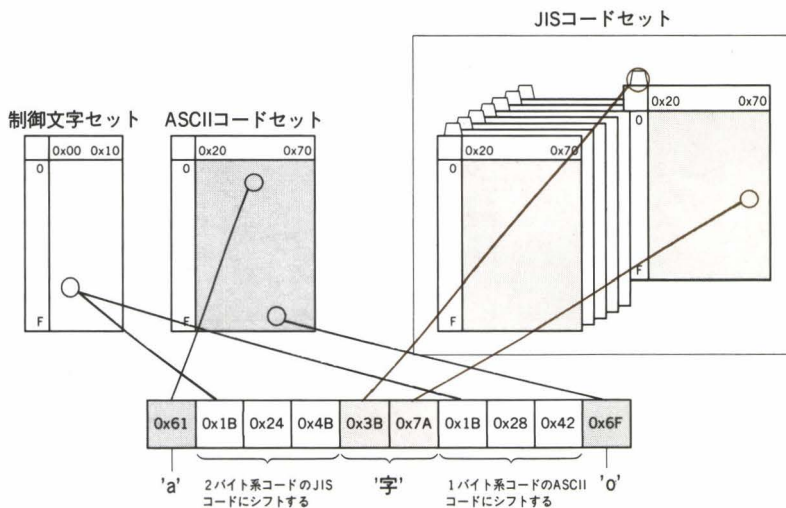



図3-2　シフトコードによる漢字の表示

さらに厳密にいうと，「JIS X 0208」には1978年に作成された「JIS X 0208-1978」と1983年に改正された「JIS X 0208-1983」が存在し，一般に前者を**旧JIS**，後者を**新JIS**と呼んでます．これらの差異についてここでは深くふれませんが，この両者は通常あまり使われない難字の割りつけが微妙に異なっています．人名など固有名詞の旧字体や異体字と呼ばれる文字を使うときには注意してください．全体的な傾向として，MS-DOSを使用するパソコンの多くが旧JISを，UNIXを使用するコンピュータ(端末として独立していない部分)のほとんどが新JISを使用しているようです．

まとめとして，これらのシフトコードを**表3-2**に示します．

文字セット	シフトコード
JIS X 0208 − 1983	ESC $ B
JIS X 0208 − 1978	ESC $ @
JIS X 0201	ESC (J
ASCII	ESC (B

(JIS X 0202より引用)

表3-2　シフトコード(新JIS／旧JIS／JISローマ字／ASCII)

■ 8ビット系漢字コード

JISコードは〝情報交換用漢字符号系〟と名づけられているように，あくまで通信のためのコード体系です．ですから，通信回線の途中に7ビットコードしか使えないようなシステムがあっても，問題

なく漢字を送受信できるように作られています．しかし，JIS コードで表された全角文字を編集するとなると，後節で解説するさまざまな問題点が出てきます．とくに問題となるのは，シフトコードによって同一のコードに複数の意味を持たせている点で，着目したコードが全角文字かそうでないかは，前後をスキャンしてシフトコードを調べないと判断がつきません．

しかし，8 ビットコードが使えるシステムで使用することを前提とすれば，編集処理がしやすいコードを作成することが可能です．とくにシフトコードを使わないコード系を用いることができれば，プログラムの開発／実行効率が大幅に向上します．

そこで各 OS ごとに，シフトコードなしで全角文字を表現できる内部処理用の 8 ビット系コードが作成され使われています．以下にこれらのコード体系について概説しておきます．

シフト JIS（Microsoft 漢字）コード

このシフト JIS コードは，現在のところ最も広く使用されている 8 ビット系漢字コードで，パーソナルコンピュータの世界では，事実上標準となっているコードです．さらに，BBS システムなどでは，内部コードとしてだけでなく通信用コードとして使用している場合もあります．

〈第1バイトのコード表〉
7ビット系コードと共通の領域
8ビット系コード固有の領域
0x00 0x10 0x20 0x70 0x80 0x90 0xA0 0xD0 0xE0 0xF0
制御文字領域
アルファベット文字領域
全角文字領域（本来は未定義領域）
カナ文字領域
全角文字領域（本来は未定義領域）
アルファベットでもカナ文字でもないコードがきたら，全角文字であるとして第2バイトを受けとる
〈第1バイトが0x8Eの第2バイトのコード表〉
0x20 0x40 0x70 0xF0
未定義領域
重複する領域 半角アルファベットと
全角文字領域
文字列……0x41 0x8E 0x9A
半角文字 A
全角文字 字
シフトコードがなくても全角文字が扱える

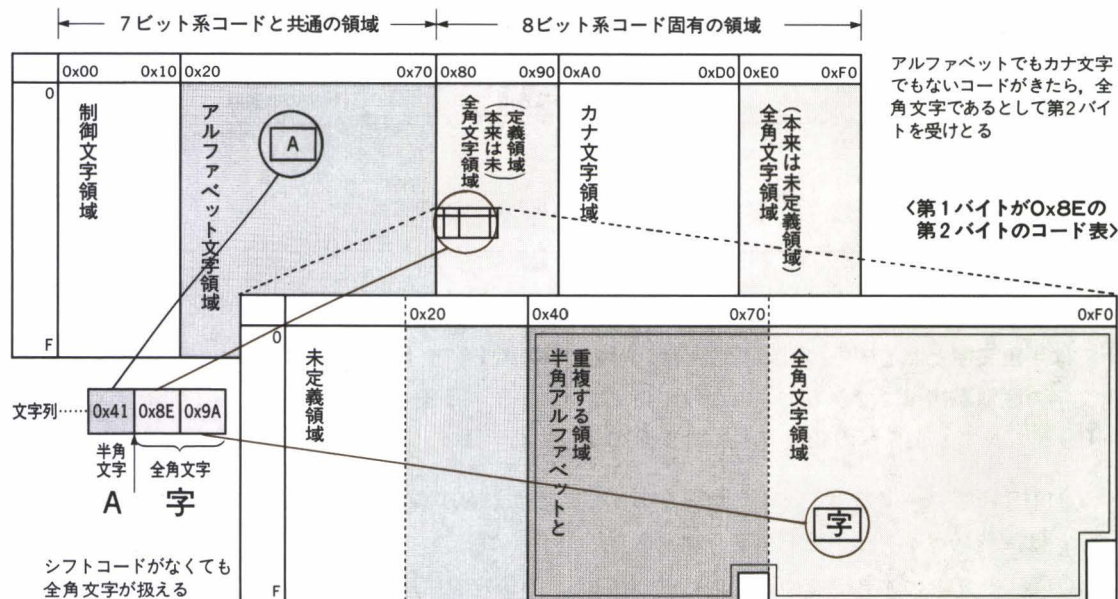



図 3-3　シフト JIS コードによる漢字の表現

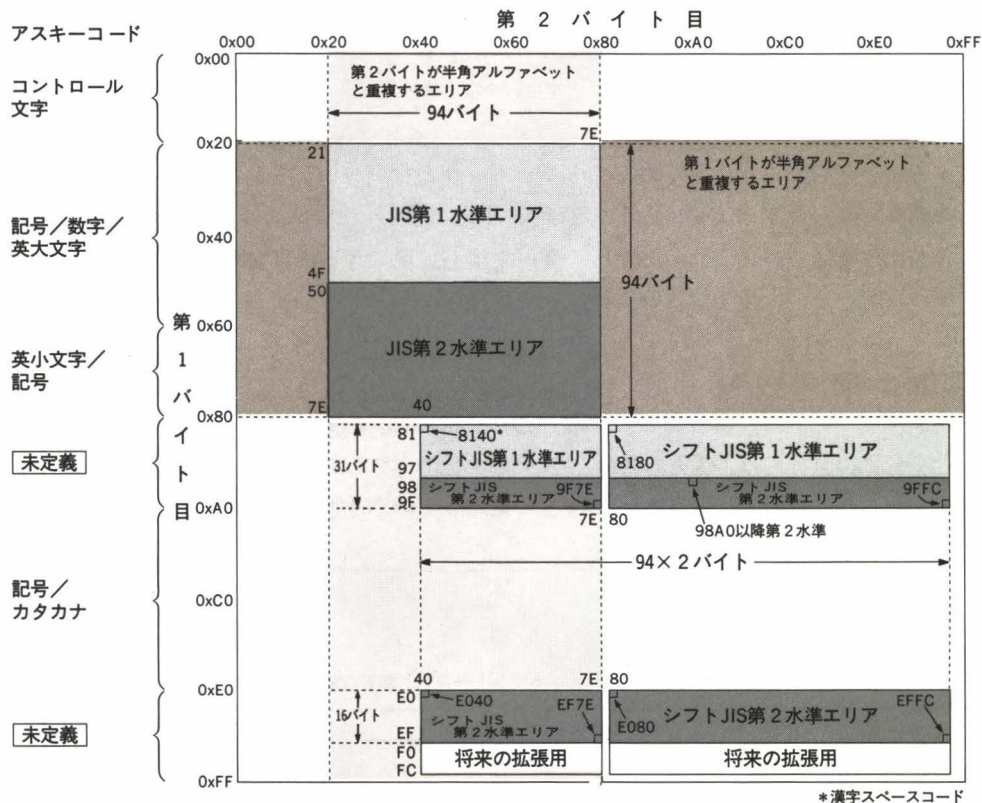



図3-4　シフトJISとJISコードの分布図

シフトJISコードは，「JIS X 0201」で定められている8ビットコード(アルファベットとカナ，以下では**半角文字**と記述する)の未定義部分に全角文字の第1バイトを割り当てることで，全角文字と半角文字の区別が簡単にできるようになっています(**図3-3**，**図3-4**を参照)．

JISからシフトJISへの変換方法は**図3-5**のように行います．

①JIS漢字コードの第1バイトが隣りあう2区どうしを1つにまとめ，第1バイトが偶数のものは，第2バイトにオフセットを加える．
②第1バイトのすき間をつめて，「JIS X 0201」の未定義領域から始まるように全体を移動する．
③DELコード(0x7F)に重ならないように第2バイトにオフセットを加える．
④半角のカナ文字(0xA0～0xDF)に重ならないように第1バイトにオフセットを加える．

① 第1バイトを2つずつに分け，第1バイトが偶数のものは1減らして第2バイトにオフセット(0x5E)を加える

② 第1バイトのすき間をつめて，2121を8140の部分に割り当てる

③ 第2バイトがDEL(0x7F)に重なるのを避ける

④ 第1バイトがカタカナのコードに重なるのを避ける

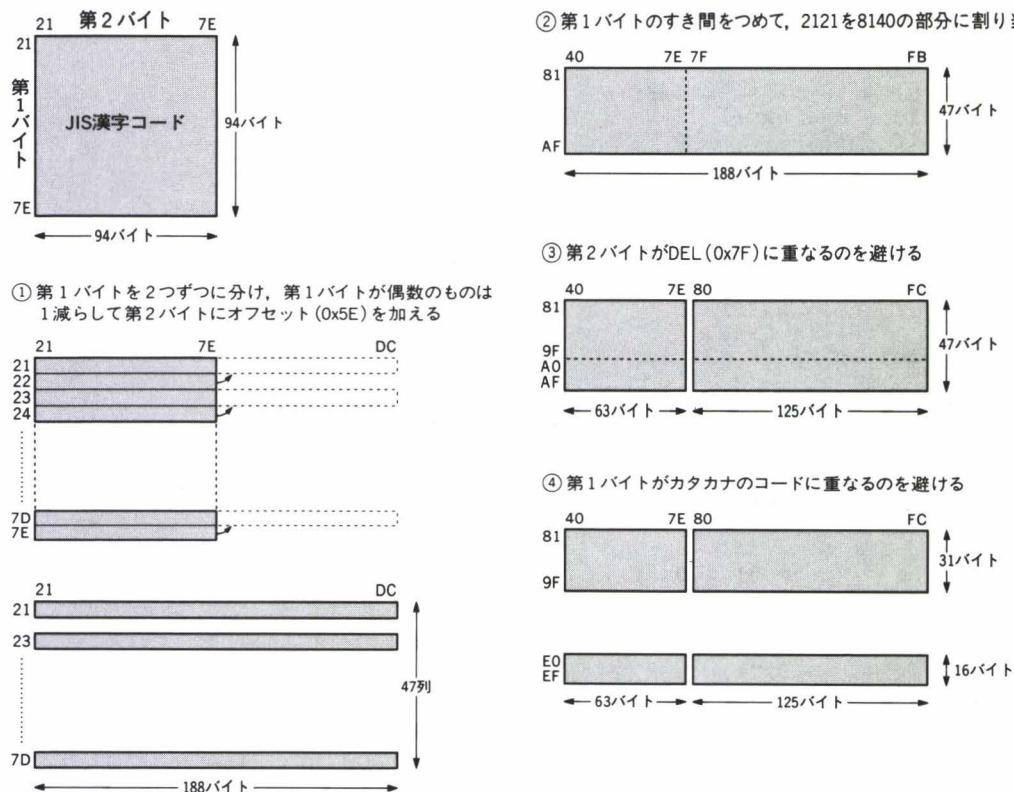



図 3-5 JIS →シフト JIS 変換手順

JIS／シフト JIS を相互に変換する C のプログラムを次ページの**リスト 3-1** に示します．

シフト JIS コードは，当然のこととして 8 ビット系コードの長所(シフトコードを使わなくてもすみ，第 1 バイトに着目すれば第 2 バイトも含めて全角文字であることが即座にわかる)をそなえています．

さらに，後述する EUC コードと違って，半角文字をそのまま使えること，全角文字の第 1 バイトと第 2 バイトの分布の重複が 30%ほどの領域に限られているため，第 1 バイトと第 2 バイトの判別がしやすいなどの長所もあります．これらの特徴から，パソコン通信のような文字化けの起きやすい通信路でも文字化けが尾を引きにくく，半角文字も完全に読み書きできるコード系として広く使われています．

```
 1: jis2sjis(int c)
 2: {
 3:     int hi, lo;
 4:
 5:     hi = (c >> 8) & 0xff;
 6:     lo = c & 0xff;
 7:     if (hi & 1)                 ┐
 8:         lo += 0x1f;             │ ①
 9:     else                        │
10:         lo += 0x7d;             ┘
11:     hi = (hi - 0x21 >> 1) + 0x81; ………… ②
12:     if (lo >= 0x7f)             ┐ ③
13:         lo++;                   ┘
14:     if (hi > 0x9f)              ┐ ④
15:         hi += 0x40;             ┘
16:     return hi << 8 | lo;
17: }
```

```
 1: sjis2jis(int c)
 2: {
 3:     int hi, lo;
 4:
 5:     hi = (c >> 8) & 0xff;
 6:     lo = c & 0xff;
 7:     hi -= (hi <= 0x9f) ? 0x71 : 0xb1; ………… ④
 8:     hi = hi * 2 + 1; ………………………… ①
 9:     if (lo > 0x7f)              ┐ ③
10:         lo--;                   ┘
11:     if (lo >= 0x9e) {           ┐
12:         lo -= 0x7d;             │
13:         hi++; …………………          │ ②
14:     } else                      │
15:         lo -= 0x1f;             ┘
16:     return hi << 8 | lo;
17: }
```

リスト 3-1　JIS_SJIS.C

しかし，前掲の図 3-4 に示すように全角文字の第 2 バイトと半角文字の分布は重なっているため，文字列を後ろからスキャンしてくるなどして全角文字の第 2 バイトに着目した場合，前のバイトを見ないと全角文字か半角文字かの判別ができず，全角文字のことを考慮していない処理系やライブラリ関数を使用した場合，全角文字の第 2 バイトと半角文字が同一視されてしまうという欠点が存在します(図 3-6)．

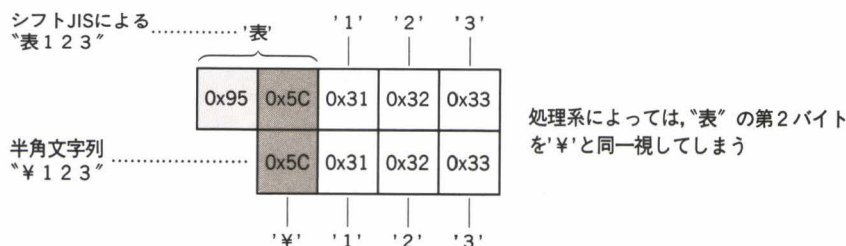



図3-6 全角文字と半角文字の重複

拡張 UNIX コード(EUC コード)

この文章を書いている1990年後半の時点では，各社が安価なUNIXワークステーションを出しはじめ，パーソナルコンピュータを使っているユーザーにもUNIXがだんだんと意識されはじめてきています．ここで問題になるのは「国際文字コード」問題です．

国際文字コードといってもピンとこないかも知れませんが，日本ではこれが「漢字コードの問題」ということになる，といえばよくわかると思います．

日本のUNIXの世界でもすでにいくつかの漢字コードを独自に決めているメーカーがあり，それらの規格はかなり違うのが現状です．

なかには国際規格体系を全く無視したようなものや，今までのように「データがシリアルに転送される」ことを前提として作ったものなどのように，現状にそぐわないものなどもあります．

UNIXの世界ではUNIXの供給元であるAT&T社が主導し，これらの国際規格を決めました．これらの規格はISO(国際標準化委員会)にも登録され，現在ISO-2022として国際的に認められています．

EUCは1つの基本コードセット，3つの補助コードセットで構成されていて，それぞれ「コードセット0」「コードセット1」「コードセット2」「コードセット3」と呼びます．基本コードセットは基本的に7ビット表現のASCIIコードと同じです．

システムの立ち上がり時にはシステムで扱うコードセットは基本コードセットで，ここから，1，2，3の各コードセットに切り替えることができます．切り替えは，コードセット0でない場合を8ビットデータの最上位ビットを立ててそれを示します．

コードセット2，3への切り替えは「SS2(0x8E)」「SS3(0x8F)」という「シングルシフト文字」によってそれぞれ行います．

すなわち，コードセット0から他のコードセットへの切り替えはまず，最上位ビットの判定で行い，次に1,2,3のどのコードセットであるかの判定はその値が0x8E(この場合はコードセット2)か0x8F

(この場合はコードセット3)かそのどれでもないか(この場合はコードセット1)，という判定で行うわけです．

コードセット1，2，3いずれの場合も，それぞれのコードセットから基本コードセットへ戻るきっかけはやはり最上位ビットの判定によります．つまり，コードセット1，2，3の場合，そうであるあ

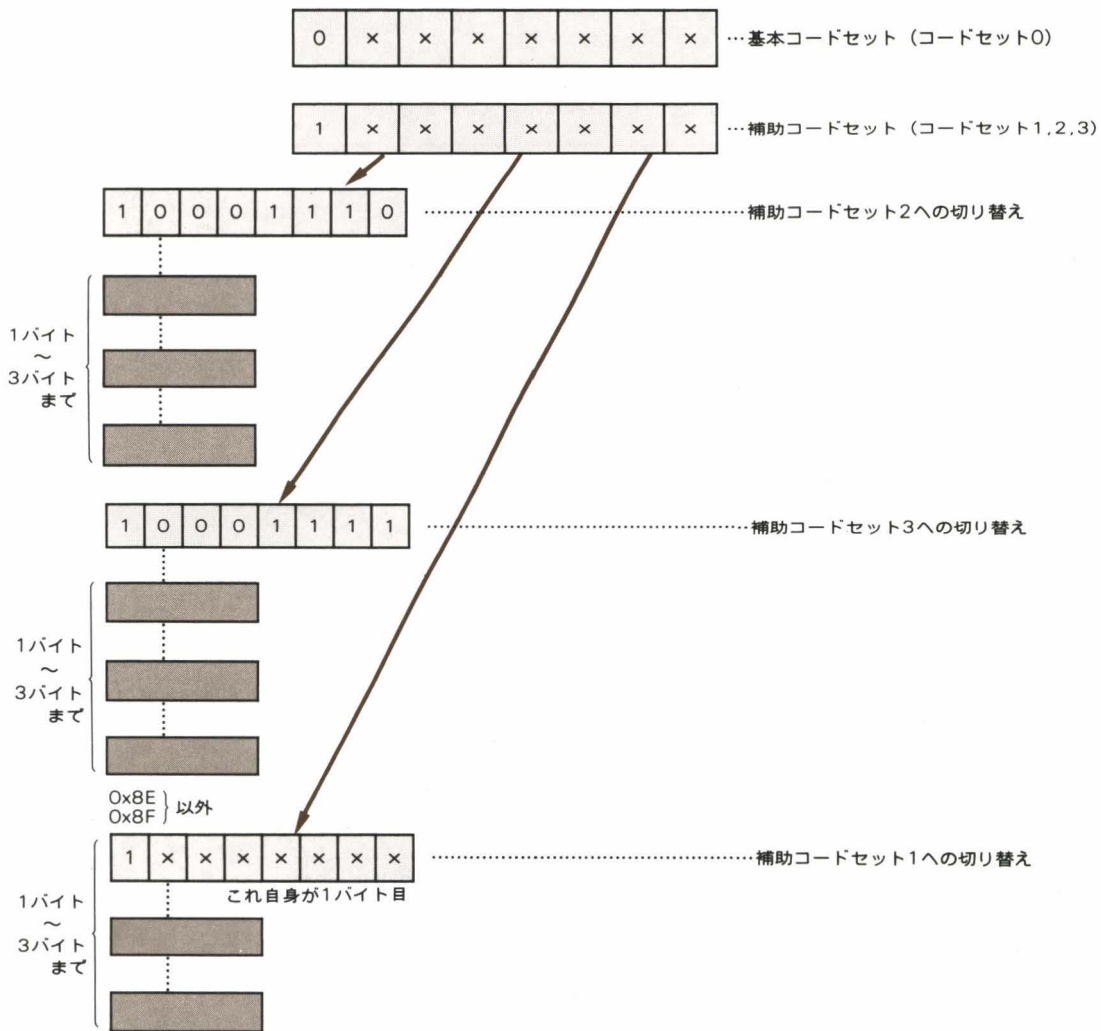



図3-7　EUCのコードセット

いだはすべての文字データの最上位ビットが立っていないといけません．

また，コードセット1，2，3のいずれも1バイト，2バイト，3バイトのコード体系を許しています．これらを図にすると**図3-7**のようになります．

また，EUCでは特殊文字として0x20（スペース），0x7F（削除），基本制御コード（0x00から0x1Fまで），補助制御コード（0x80から0x9Fまで）を許しています．これらの文字はコードセットを切り替えるなどの制御を行うことになっています．これらの規格はISO-2022，ISO-6937/3によって規定されている通りです．

日本ではJIS-X0208（全角漢字），JIS-X0201（半角カナ）で示されている漢字コードをそれぞれコードセット1，コードセット2で定義し，コードセット3にはユーザー定義の，いわゆる「外字」を割り当てて使うのが一般的になっています．

この拡張UNIXコードは拡張性が高く，また現代のコンピュータにマッチしたコード体系なのですが，プログラムを組む側からいうとマルチバイト文字（1バイトよりも大きいバイト数で1文字を表現する文字）が何バイトになるのか，という規定がなく，また国や言語別の固定化された識別子を持たないために，よりいっそうの国際文字コードの混乱を招く恐れがある，という面があることも現時点では事実です．

■ コードの互換性

前述の2つの8ビット系コードは，あくまで内部処理用のコードですから，外部との通信やデータ交換の際には，JISコードに変換しなければなりません（どれもJISコードをなんらかの方法でシフトしたものなので，比較的簡単にJISコードに変換することができる）．

また，同じコード体系に準拠している場合でも，同一のデータで異なる文字が表示される場合があります．たとえば同じシフトJISでも，NECのPC-9801上のMS-DOSでは「JIS X 0208」からのシフトではなく，PC-9801に搭載されているNEC漢字からのシフトになります．NEC漢字は，基本的な部分は旧JISと同一ですが，縦線や横棒などの罫線素片やローマ数字，2バイト半角文字などを独自に割り当てています．そのため，この独自のコードを使用すると，他機種のパソコンでは正しく表示されません．

また同様の問題は，新JISから変換するのか旧JISから変換するのかについても起こります．同一のシフトJISコードやEUCコードを出力しても，新／旧JISのどちら側の文字が表示されるのかは，CRT端末やプリンタに依存することになります．MS-DOSマシンの多くは，旧JIS仕様の文字フォントを搭載していますから，MS-DOS上でシフトJISコードの文字列を表示した場合，旧JISに準拠した文字で表示されることになります．

3.2 プログラムソース中の全角文字の扱い

本章の最初で述べたように，米国では7ビットしか使わないASCIIコードを使用しています．今のところ，C言語の処理系のほとんどが米国製ですから，8ビットコードはかならずしも通らない場合があります．また8ビットコードが使用可能な処理系でも，多バイトで1文字を表すことを前提としていない処理系に，最上位ビットを通すように改造を施しただけのことが多く，一見バグになりそうもないところでバグが発生してとまどうこともあります．本節ではコンパイル時に発生する問題について解説していきます．

■ 8ビットコードを使用できる処理系

処理系が8ビットコードに対応していると，全角文字の扱いはかなり楽になります．しかし，シフトJISを使っている場合，76ページの図3-4で示したように，全角文字の第2バイトの分布が半角アルファベットの一部(0x40〜0xFC)と重なっているために問題が生じることがあります(**図3-8**)．

```
C>type noesc.c
main()
{
    printf("表示¥n");
}          0x95 0x5C 0x8E 0xA6
                "¥"        "ヲ"
C>noesc          無視されてしまう
侮f
    0x95 0x8E 0xA6
C>
```

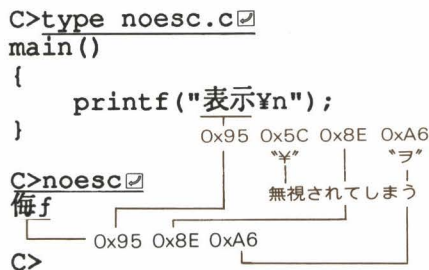



図3-8　「¥」による影響(シフトJIS)

シフトJISの第2バイトとアルファベットの重複のうち，コンパイル時に問題となるのは「¥」のみですから，全角文字列中の「¥」を「¥¥」に変換することで，とりあえず問題は解決されます．以前は日本語対応の処理系の多くが，プリプロセッサでこの処理をしていました(**図3-9**)．

```
C>type bsr.c
main()
{
    printf("表¥示¥n");
}
           0x95 0x5C 0x5C 0x8E 0xA6
C>bsr        "¥"  "¥"
表示        コンパイラに0x5Cコードとして
            解釈される
C>
```

図 3-9　「¥」による影響への対策(シフト JIS)

最近の日本語対応コンパイラは，コンパイル時にオプションを指定することなどでコンパイル中に同時に変換してくれるものが多くなってきました．

しかしながら，日本語対応といっても，char 型で 1 バイトの文字しか想定していない現状では，「c == '字'」という記述は当然できません．そのため，将来的には long char 型のサポートのアナウンスがあることもあり，全角文字 1 文字を 1 つの単位として扱える処理系の登場が待たれています．

■ 7ビットコードのみ使用できる処理系

MS-DOS 上で 8 ビットコードが使用可能な処理系を使う場合には，前述のようにソースコード中に全角文字を埋め込むことはそれほど問題になりません．それに対して，一部の UNIX などのように，7 ビット系コードを使わざるを得ないような環境でプログラムを作成する場合，そのままではプログラムのコンパイルができない場合が数多く存在します．というのも，コンパイラの側では，シフトコードを ESC に続く単なる文字列としか見ないために，全角文字列の定数中に「 " 」(0x22)や「¥」(0x5C)などが出現している可能性があるためです．

たとえば，全角文字列を構成している第 1，第 2 バイトのどちらかに「 " 」が存在した場合，そこで文字列定数がいったん切れてしまい，「 " 」以降の文字列がシンタックスエラーを引き起こすことになります(図 3-10)．また，文字列中に「¥」(0x5C)が存在した場合は，その後ろの文字がエスケープされてしまうことになり，実行時の出力結果が意図したものと異なってしまうばかりか，シフトコードが欠けて，プログラムの終了後にも影響を及ぼしかねません(図 3-11)．

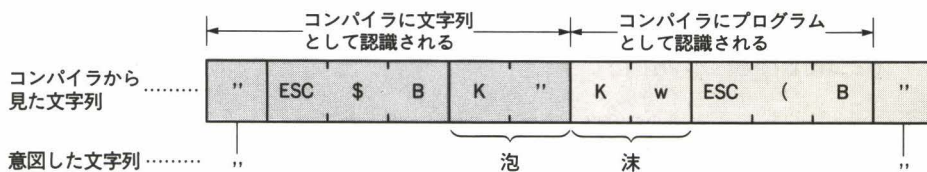



図3-10　「"」による影響(JIS)

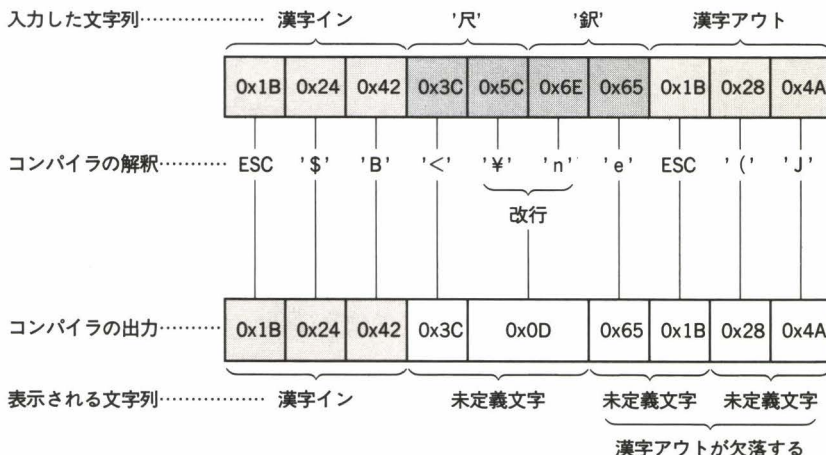



図3-11　「¥」による影響(JIS)

そのため，シフトコードの扱いなどを含めたプリプロセッサを自分で作成する必要が出てきます．しかし，そのような労力をかけずに，せめてエラーメッセージだけでも日本語にしたいということがあるでしょう．そこで，メッセージを別ファイルに収めるという方法を次ページの**リスト3-2**に紹介します．この方法であれば，日本語のメッセージであっても別ファイルに収められているので，コンパイル時に「¥」がエスケープされるなどの問題がありません．さらにユーザーの側でメッセージに手を加えて，システムを使いやすくしていくこともできるようになります．

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <string.h>
 4:
 5: #define PTR_MAX 20……………メッセージの個数
 6: #define STR_LEN 256…………メッセージの最大長
 7:
 8: extern  unsigned char *msg[]; …………………メッセージへのポインタの配列
 9: extern  unsigned char msg_file[];…………メッセージファイルのファイル名
10:
11: read_msg()
12: {
13:     unsigned char buff[STR_LEN];
14:     FILE  *fp;
15:     int   i;
16:
17:     i = 0;
18:     if ((fp = fopen(msg_file, "r")) == NULL){
19:     perror("Message file");
20:     exit(-1);
21:     }
22:     while (fgets(buff, STR_LEN, fp) != NULL){
23:     if (i > PTR_MAX){
24:         fprintf(stderr, "Warning : Too many messages.¥n");
25:         return;
26:     }
27:     if ((msg[i] = malloc(strlen(buff))) == NULL){
28:         fprintf(stderr, "Can't get memory : message storage area¥n");
29:         exit(-1);
30:     }
31:     strcpy(msg[i++], buff);
32:     }
33: }
```

リスト 3-2　READMESG.C

この場合，気をつけなければならないことは，プログラム作成時に，メッセージファイルの何行目のメッセージが，どのソースファイル中のどのポインタに対応しているかわからなくなってしまう可能性がある点です．メッセージファイル中のメッセージとポインタの対応がずれると，まったく意図しないところで関係のないメッセージが出力されてしまうことになります．

そこで，そのような事故を防ぐために，プログラムを組む際には，ソースファイルにメッセージをそのまま書いておくことにします．そのメッセージがはいったままのソースファイルを，コンパイルの直前にメッセージの抽出とポインタへの置き換えをするプリプロセッサで処理するようにすると，プログラム開発の効率が上がるでしょう．

3.3 全角文字の入出力

文字列の入出力の部分での問題は，後節で解説する編集作業に比べると少ないといえるでしょう．内部処理を簡単にするために内部文字コードを独自に設けて作業をしている場合には，入出力の部分に負担がかかりますが，そのような場合を除けば，ライブラリ関数をそのまま使用できることが多く，たとえputchar関数が8ビットコードを通さなかったとしても，write関数で代用するなどの工夫で解決できる場合がほとんどです．

とりあえず，以降で解説するような事項に注意を払えば問題は起きないでしょう．

■ 全角文字の制御

1文字を2バイトで表すという方法は，日本国内でも10年ほどまえからやっと使われ始めた方法で，全体の歴史が浅いコンピュータの世界でも比較的目新しい手法といってよいでしょう．そのため，日本語対応のコンピュータでさえ，全角文字を半角文字と同じ方法で出力すると，いくつかの問題が生じてしまいます．

たとえば，端末やプリンタなどの多くは，印字位置の制御を半角文字を基準に行っています．そこで，80桁表示可能な画面で79桁目に全角文字を出力する場合の処理は，機器によって異なったり，表示が崩れてその後の出力に影響を及ぼしてしまうこともあります．

そのため，全角文字をCRT端末やプリンタに出力する際には，**リスト3-3**に示すように文字数を数えて，「＜表示可能幅＞－1」の位置には全角文字を送らないようにする対策をとらなければなりません．

```
119:            if (col >= Len - 1) {      /*  ? col is limit   */
120:                if (Kflag && kintbl (c1, c2)) {     /*  ? kinsoku   */
121:                    putchar (c1);
122:                    putchar (c2);
123:                    col += 2;
124:                    continue;
125:                }
126:                putchar ('¥n');
127:                col = 0;
128:            }
129:            putchar (c1);
130:            putchar (c2);
131:            col += 2;
```

（注記：120行 if → 出力桁数を調べている／123行 → 桁数の計算／126～127行 → 表示する桁数に達するまえに改行する）

リスト3-3　表示幅による制御(JFOLD.C ＜シフトJIS版＞より抜粋)

この問題は，端末側(CRT ドライバやプリンタなど)が行う改行の問題ですが，プログラムの側で行う改行についても問題になります．全角文字の第1バイトのあと，第2バイトを出力するまえに，改行やスペース，タブ，バックスペースなどを出力すると，文字が正常に出力されなくなることがあります．また特殊な例としては，画面制御関数でカーソルを操作しているプログラムが，全角文字の右半分や左半分に文字を重ねてしまったら，その後の表示はどうなるかという問題もあります．

このような問題から，画面制御(改行なども含む)をする際には，かならず〝1文字単位〃で画面制御をしなければなりません．とくに，全角文字と半角文字が混在している場合には，半角文字がどこにどれだけ存在しているのか，画面全体のことを考えておく必要があります．

問題が発生するのは出力だけにかぎりません．内部コードと OS 上のコードの変換を行う場合など，自分でバッファリングを行うような関数を作成する際にも，気をつけなければならない点が存在します．そのような関数で文字列を順次加工していくと，たとえば**図 3-12** のように途中でバッファの境界に全角文字の第1バイトと第2バイトがまたがってしまうことがあります．こういった問題を避ける最も単純な方法は，バッファリングを行わず，入出力をかならず文字単位で行うようにするという消極的な手をとることです．

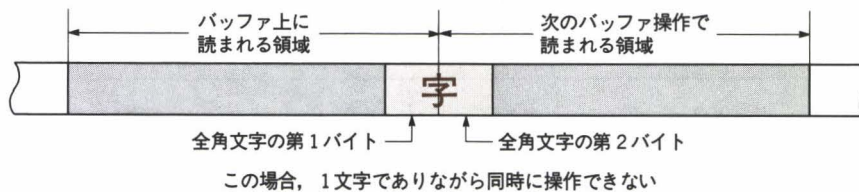



図 3-12　バッファ境界上の全角文字の問題

しかし，実行速度を下げたくないなどの理由で，どうしてもバッファリングを行いたいならば，バッファの境界上に全角文字がまたがってしまったときに特殊なバッファ操作を行います．その場合は，バッファリングするサイズを一時的に変更して，加工し終わった文字列のみを出力します．中途半端な残りの第1バイトの加工については，次の読み込みまで加工を保留して，次のバッファの先頭にあるデータと合わせて処理をします．

■ 7ビット系コードの入出力処理

プログラムが7ビットコードを使って全角文字を表示する場合，前項で挙げた点に加えて，文字セットの切り替え(全角モードと半角モード)についても対策をとらなければなりません．

今のところ，全角文字列を出力している最中に，全角モードを抜けずに制御文字(改行文字など)を出力した場合，正常に表示できなくなってしまうデバイスが数多く存在します．そのため，改行を送るまえにかならず半角モードに移っておく必要があります(**リスト3-4**)．

```
156:            if (col >= Len - 1) {     /*  ? col is limit   */
157:                if (Kflag && kintbl (c1, c2)) {     /*  ? kinsoku    */
158:                    putchar (c1);
159:                    putchar (c2);
160:                    col += 2;
161:                    continue;
162:                }
163:                putchar (ESC);         } 全角モードを一度抜ける
164:                fputs (Ko, stdout);    }
165:                putchar ('¥n'); ..........改行を出力
166:                putchar (ESC);         } 全角モードに入りなおす
167:                fputs (Ki, stdout);    }
168:                col = 0;
169:            }
170:            putchar (c1);
171:            putchar (c2);
172:            col += 2;
```

リスト3-4　全角モード中の改行動作(JFOLD2.C <JIS版>より抜粋)

同時に，文字セットの切り替えには，コードの重複という問題が常につきまといます．JISコードの場合，全角文字の第1／第2バイトの分布と「！」(0x21)から「~」(0x7E)までのすべての半角文字の分布が重なりますから，シフトJISでは問題にならない「%」記号が，printfなどの引数中に存在することになります．

こういった問題が生じることから，全角文字列の解析は自前の関数で処理するのが原則です．しかし，どうしても全角文字列をライブラリ関数で処理しなければならないときには，関数の呼び出しのまえに，引数となる文字列を解析し，その内容に応じた前処理を行います．

シフトコードについても気をつけなければならない点があります．全角文字列を入力中に，一度漢字入力モードを抜けたものの，もう一度漢字入力モードにはいりなおして文字列を入力した場合，次ページの**図3-13**に示すように，表示にまったく影響を与えない**漢字アウト**(以下KOと記述)／**漢字イン**(以下KIと記述)の組み合わせが発生することになります．

また，端末やかな漢字変換フロントエンドプロセッサの構成によっては，かな漢字変換のたびに「KI/KO」や「KO/KI」が繰り返されることもあり，表示にまったく関係のないデータが大量に格納領域を占有することになります．データの格納効率を考えると，このようなむだな「KI/KO」は取り除かなければなりません．

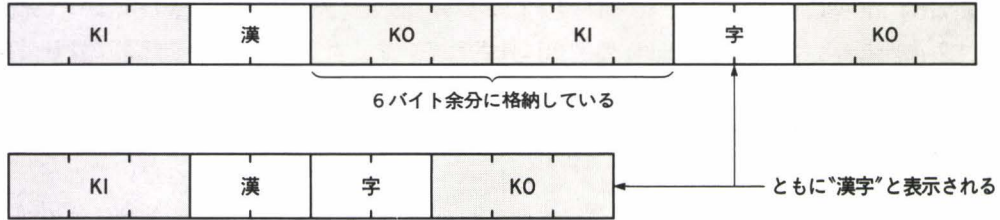



図3-13　むだなシフトコードによってデータがふくれあがる例

ただし，パソコン通信など，公衆電話回線のような文字化けの起きやすい信頼性の低い通信路を使用する場合には，話が逆になってきます．7ビット系コードを使用しているときに，KIコードやKOコードが欠落した場合，その後の表示は意味をなさなくなってしまいます(**図3-14**)．このような事故が起きないようにするには，頻繁にKI/KOコードを送ってやるのが最も手軽で確実です．

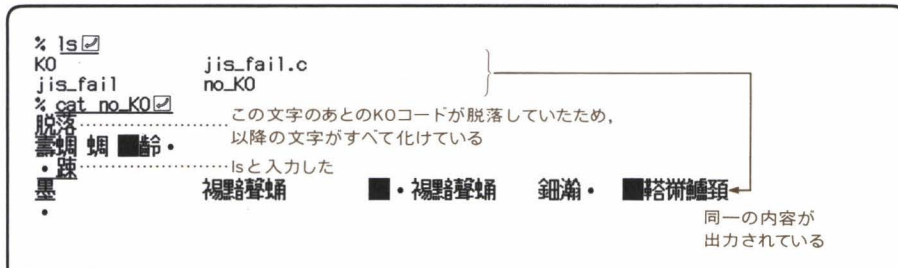



図3-14　KOコードが脱落した場合の表示

そのほかにも，対話的な処理を行っているときに，文字列の途中で出力を止めると，KOコードが送られていないために端末へのエコーバックが化けてしまうことがあります．また，出力した文字数と実際に表示される文字数がシフトコードの関係でずれますし，シフトコードとデバイス制御のためのエスケープシーケンスとの区別が2，3バイト先読みしなければわからないなど，実際にプログラムを作成していくとたくさんの問題点がでてきます．

残念ながら，これらの事例に対して，こうすればかならず大丈夫という対策を示すことはできません．しかし，それぞれのプログラムのアルゴリズムと，発生する問題点に応じた対策はかならず存在します．根気よく対策を練っていけば解決策が見つかるでしょう．

■ 8ビット系コードの入出力処理

いうまでもなく，8ビットコードでは8ビット目が立っているコードが存在しています．C言語では，char型はint型に拡張されたうえで操作されますが，拡張に際して8ビット目によって正負に符号拡張するのかしないのかについては，処理系に任されていました（ANSIの標準化案では符号拡張することになっている）．

そのため，シフトJISの第1バイトを判断する場合などに，「(c>0x80 && c<0xa0)」などとすると，cの中身が符号拡張されて0xFF81などになっていて，判断が正常に行われないことがあるので注意が必要です．また，配列の添え字に符号拡張された文字変数を使用すると，配列領域の外を参照することになり，プログラムの暴走を引き起こすことにもなりかねません．

符号拡張の問題は，意外に気づかずに対策を忘れがちな問題点の1つですが，全角文字を扱う場合には，**リスト3-5**のように0xFFでかならずマスクをかけるか，char型を使用せずにunsigned char型を使用するようにします．

なお，日本語対策がとられている処理系では，日本語オプションを指定した場合，char型が自動的にunsigned char型に変換されるものがあります．

```
1: #define _iskanji(c)  (((c) & 0xff) >= 0x81 && ((c) & 0xff) <= 0x9f ||
2:                        ((c) & 0xff) >= 0xe0 && ((c) & 0xff) <= 0xfc)
                              └→符号拡張にそなえ，マスクをかけている
```

リスト3-5　_iskanji関数

また，8ビット系コードでも，エディタなどのように表示位置を厳密に求めなければならないプログラムでは，文字を表示するまえに文字列を調べて，その文字がどこに表示されるかを確かめなければなりません．さらに，8ビット系コードの使用には，ライブラリ関数が8ビットコードを使用できないという可能性がつきまとうことになります．8ビットコードを通してくれるかどうか心配ならば，バイナリデータでも通してくれるread，writeなどの低水準入出力関数を使うといいでしょう．

3.4 全角文字列の編集

前節で全角文字を入出力する際に気をつけなければならない点をいくつか挙げました．しかし，全角文字列を編集するとなると，さらにやっかいな問題がでてきます．とくに7ビット系コードでは全角文字の編集はほとんど不可能であるといってもよいでしょう(本節の前半でくわしく解説する)．そこで本節では，8ビットコードによる全角文字列の編集作業，とくに編集の前処理の部分を重点的に解説します．

なお，本節の最後に，内部コードの作成と利用について触れています．7ビットコードを使わなければならない方は，そちらも参照してください．

■ シフトコードを使用することによる問題

文字列を編集する場合は，かならず文字単位で編集しなければなりません．とくに全角文字を編集するときには2バイトずつ組みにして編集する必要があります．ところが，JIS コードのようなシフトコードを使用する文字系の場合，文字列中のある文字が全角／半角文字のどちらの文字セットに属するのか即座に判断できません．

また，JIS コードを使う場合，文字列のコピーという作業が極端に増えてしまいます．たとえば，図3-15のような場合には，半角文字列のときには必要のなかった文字列のコピー作業をしなければなりません．

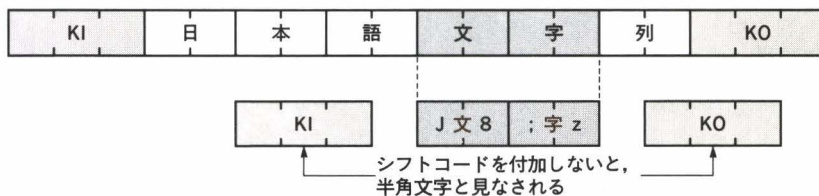



図3-15　文字列の切り取りとシフトコード

上の例はシフトコードをつけ加えなければならない場合ですが，シフトコードを取り去らなければならない例もあります．たとえば，全角文字列中に全角文字列を挿入した場合，全角モード中にさらにKIコードが現れたり，図3-16のように全角文字列の途中にKOコードがまぎれ込む可能性があります．

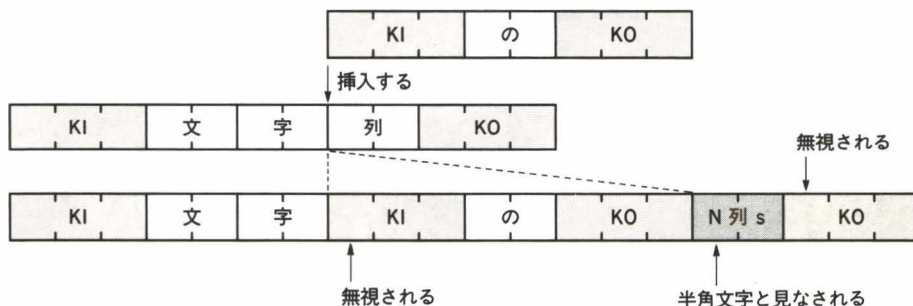



図3-16　全角文字列中に全角文字列を挿入する

シフトコードは，入れ子になっているような構造にはできません．つまり，図3-16の例では，2個目のKIコードは無視され，挿入した文字列側のKOコードによって，その後ろの文字列は半角文字列と判断されてしまいます．

これらの問題点は，そのいずれもがシフトコードを使用することによる弊害であることはおわかりだと思います．このようにシフトコードを使用するコード系は，通信に使用する場合，いろいろな利点(従来のコード系と完全にコンパチビリティを保つことができるなど)がありますが，文字列の編集作業を行うといった「作り手側のこと」になると，面倒な問題が多くなってきます．

そこで，多くのOSでは，内部処理用コードとして，シフトコードを使わなくてもすむ日本語文字コード体系を採用するようになりました．3.1節で解説した8ビット系コードは，以上のような経緯で開発され，現在にいたっているものです．

■ 編集の前処理

全角文字列を編集する場合，文字と文字の境界を求めておかなければ，文字を切り離すこともつなぐこともできません．また，半角文字と全角文字が混在している文字列を編集する場合には，全角か半角かの判断もしなければなりません．そこで，編集の前処理の部分について解説していきましょう．

8ビット系コードは7ビットコードと比較して，シフトコードを使用しなくてすむというほかに，全角／半角の判別が行いやすいという利点もそなえています．たとえば次ページの図3-17に示すように，EUC漢字コードでは8ビット目が立っているかどうかをみるだけで判断することが可能です．

また，半角カナ文字が含まれているEUCや，全角文字の第2バイトが半角アルファベットの一部と重なっているシフトJISでも，その文字か，その文字の1バイト前の文字を調べることで全角／半角の判断が可能です(図3-18)．

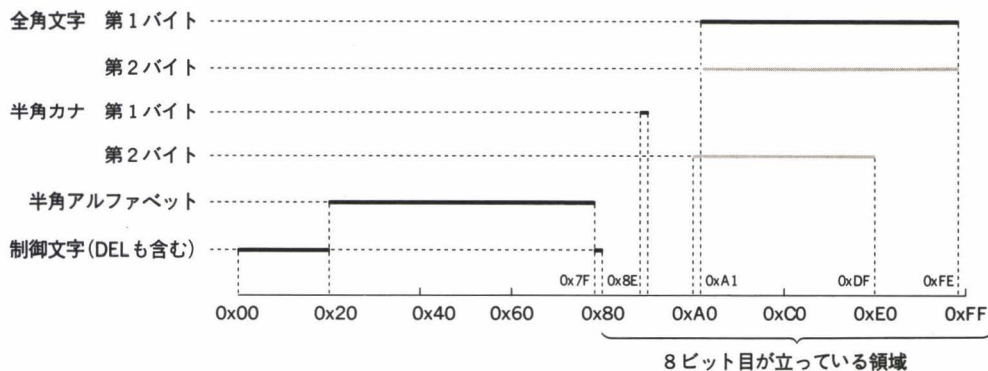



図 3-17 EUC 漢字コードの全角／半角の判断

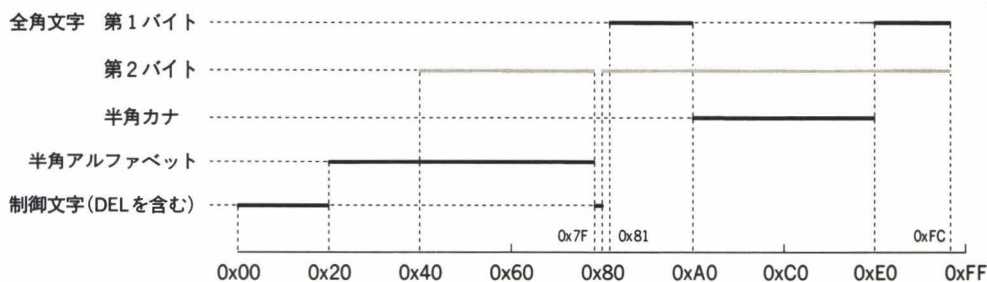



図 3-18 シフト JIS コードの全角／半角の判断

なお，前節で述べたとおり，8ビット系漢字コードには全角文字の符号拡張という問題があります．そのため，プログラミングの際，全角／半角の判断は，最もバグを出しやすい(移植性という点でも問題になりやすい)部分の1つになっていますから十分に気をつけてください．

全角文字かどうかを判断したあとには，その文字が第1バイトか第2バイトかの判断をすることになります．しかし，どの8ビット系コードでも，1文字注目しただけでは，その文字が全角の第1バイトなのか第2バイトなのか判断ができません．そこで，**図 3-19** の<a>のように，全角文字列がどこから始まっているかを調べて第1バイト／第2バイトの判別をします．シフト JIS の場合には，第1バイトと第2バイトの分布の重なりが一部分に限られているため，<b>のように全角文字列の始まりまでスキャンしなくても判別できることがあります．

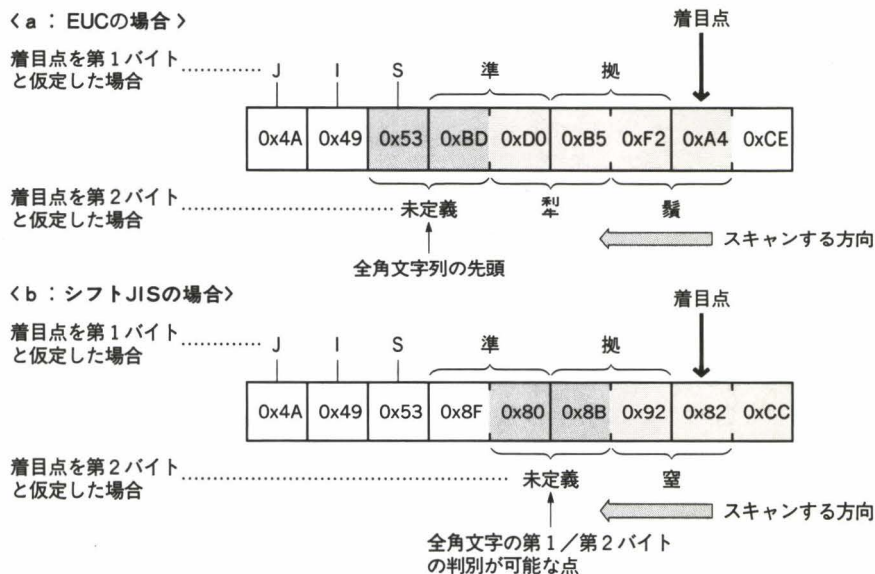



図3-19　全角文字の第1バイト／第2バイトの判断

なお，ここで忘れがちなのは，ポインタで文字列をたどっていくと，文字列の先頭を通り過ぎてしまう可能性があることで，そのようなことがないように，文字列の先頭の位置と比較しながら処理を行います．このように，全角文字に出会うたびに文字列をさかのぼっていったのでは処理効率に大きく影響してしまい，実用的なプログラムを作る際にはなんらかの工夫が必要です．

そこで，全角文字列編集の高速化手法の一例として，文字種判断をまとめて実行するようにした，全角文字列(シフトJIS)のバイナリサーチプログラムを次ページの**リスト3-6**に紹介します．

このプログラムは，第1引数に検索される文字列，第2引数に検索する文字列を渡すと，バイナリサーチのアルゴリズムで検索を行います．検索の際には，全角文字／半角文字の判断を行っていますから，あとで紹介するライブラリ関数を使った検索と違って，シフトJISコードの2バイト目が半角文字と見なされてしまうことがありません．

リスト3-6では，文字種判断の効率を向上させるために，文字列を渡された時点であらかじめ文字種を調べています(**リスト3-7**)．こうしておけば，全角文字に突き当たるたびに第1バイト／第2バイトを判断するのに比べて，はるかに効率がよくなります．また，文字種の判断をする際，文字列を終端の方向へスキャンしていますから，文字列の先頭を気にする必要がなくなっています．

```
 1: /*
 2:  *   jbsearch - seach string by binary search method
 3:  *
 4:  *     Written by Yoshiyuki Ito
 5:  *
 6:  *     Copyright (C) 1987 by Yoshiyuki Ito
 7:  *
 8:  *     Created    Nov.  1, 1987
 9:  *     Modifyed   Sep. 25, 1990   by Nishi Katsuhiro
10:  */
11:
12: #include  <stdio.h>
13: #include  <stdlib.h>
14: #include  <string.h>
15: #include  <ctype.h>
16:
17: #define ARGV0            /*  argv[0] is valid  */
18:
19: #define DEF_LEN      256
20: #define CT_ANK       0
21: #define CT_KJ1       1
22: #define CT_KJ2       2
23: #define CT_ILGL      0xff
24:
25: #define _iskanji(c)  (((c) & 0xff)>=0x81 && ((c) & 0xff)<=0x9f || ¥
26:                      ((c) & 0xff)>=0xe0 && ((c) & 0xff)<=0xfc)
27: #define chkjtype(c)  ((_iskanji((c))?CT_KJ1:((c)!='¥0'?CT_ANK:
    CT_ILGL)))
                                  └──→cの文字種を調べる
28:
29: char    *Prgnam = "jbsear";
30: char    _ch_type[DEF_LEN];
31:
32: char *jbsearch(char *s1, char *s2)
33: {
34:     int     i;
35:     int     s1len, s2len;
36:     int     iskanji1;
37:     int     p1, p2;
38:     char    *ret_p;
39:     char    *ch_type;
40:
41:     iskanji1 = 0;
42:     ret_p = NULL;
43:
44:     s1len = strlen(s1);
45:     s2len = strlen(s2);
46:
47:     if (s1len > DEF_LEN) {                                          ┐
48:         if ((ch_type = (char *)malloc(s1len+1)) == NULL){            │
49:             fprintf(stderr, "Can't get memory : jbsearch¥n");         │ 文字種を格
50:             return NULL;                                             │ 納する領域
51:         }                                                            │ の確保
52:     } else {                                                         │
53:         ch_type = _ch_type;                                          │
54:     }                                                                ┘
```

```
 55:
 56:     for (i = 0; i < s1len; ++i)
 57:         if (iskanji1){
 58:             iskanji1 = 0;
 59:             ch_type[i] = CT_KJ2;
 60:         } else {
 61:             if ((ch_type[i] = chkjtype(s1[i])) == CT_KJ1)
 62:                 iskanji1 = -1;
 63:         }
 64:
 65:     p1 = 0;
 66:     p2 = s1len-1;
 67:     if (ch_type[p2] == CT_KJ2)
 68:         --p2;
 69:
 70:     while (p1 <= p2){
 71:         if (strncmp(s1+p1, s2, s2len) == 0){
 72:             ret_p = s1+p1;
 73:             break;
 74:         }
 75:         if (strncmp(s1+p2, s2, s2len) == 0){
 76:             ret_p = s1+p2;
 77:             break;
 78:         }
 79:         if (ch_type[p1] == CT_KJ1)
 80:             p1 += 2;
 81:         else
 82:             ++p1;
 83:         if (ch_type[p2-1] == CT_KJ2)
 84:             p2 -= 2;
 85:         else
 86:             --p2;
 87:     }
 88:     if (s1len > DEF_LEN)
 89:         free(ch_type);
 90:
 91:     return ret_p;
 92: }
 93:
 94: main(int argc, char **argv)
 95: {
 96:     char *mp;
 97:
 98: #ifdef  ARGV0
 99:     Prgnam = *argv;
100: #endif
101:     if (argc != 3){
102:         fputs("Usage: ", stderr);
103:         fputs(Prgnam, stderr);
104:         fputs(" <string> <search word>¥n", stderr);
105:         exit(2);
106:     }
107:     if ((mp = jbsearch(argv[1], argv[2])) == NULL){
108:         puts("No match¥n");
109:         exit(0);
110:     } else {
```

注釈：
- 56〜63行：文字種の判別
- 70行：バイナリサーチの実行
- 79〜86行：全角文字は，２バイト単位で扱う（全角文字の第２バイトと半角文字を混同せずにすむ）
- 107行 argv[1]：第１引数に被検索文字列
- 107行 argv[2]：第２引数に検索文字列を渡す（全角文字も渡せる）

```
111:            puts(mp);
112:            exit(1);
113:        }
114: }
```

リスト 3-6　JBSEARCH.C

```
56:     for (i = 0; i < s1len; ++i)
57:         if (iskanji1){
58:             iskanji1 = 0;
59:             ch_type[i] = CT_KJ2;
60:         } else {
61:             if ((ch_type[i] = chkjtype(s1[i])) == CT_KJ1)
62:                 iskanji1 = -1;
63:         }
```

全角文字の第2バイトと半角文字の判別ができないので，第1バイトがきたら，無条件に判断している

文字列中の文字種を格納していく

リスト 3-7　文字列のスキャン(JBSEARCH.C より抜粋)

なお，エディタのような頻繁に文字列を編集するプログラムでは，これでも効率がよくないと感じられることがあります．その場合には，文字列を格納している領域と文字種を格納している領域を，構造体にするなどして同時に操作するようにすれば，さらに効率が改善されるでしょう．

■ 全角文字列編集用のライブラリ関数

現在のところ，いままでの処理系と互換性を保つ必要があることから，文字列の編集を行うライブラリ関数に日本語対策がとられている処理系はほとんどないようです．また日本語対応の処理系でも，日本語編集用のライブラリ関数は，半角文字用のライブラリ関数とは別に独自の名前で用意されています．

以下では，従来のライブラリ関数をそのまま使用した場合に生じる問題点を述べますが，いずれの場合も，新たな関数を作成する以外に対処方法がありません．そこで解決策は，日本語対応の関数名を MS-C を例に紹介するにとどめます．MS-C や Lattice C 以外の処理系を利用する場合には，相当する日本語処理関数を探すか，自分で関数を作成する必要があります．

文字列をスキャンする関数

文字列をスキャンする関数をシフト JIS の全角文字列に対して実行すると，**図 3-20** に示すように，シフト JIS の第2バイトと半角文字の判別ができずに，異常な結果を返す場合があります．

また，〝引数文字列中のどれか1文字〞といった指定を行う関数では，全角文字の第1バイトと第2バイトを別々の文字として認識してしまうことがあります(**図 3-21**)．

```
C>type t_chr.c
#include <stdio.h>

extern          char *strchr();

main()
{
     puts(strchr("シフトJISにVがある",'V'));
}
                       ここへのポインタが返されることを期待している
C>t_chr
Vフ下JISにVがある
    └── "シ"の第2バイトが"V"であるため，期待した結果が得られない
C>
```

図3-20　全角文字列での半角文字の検索

```
C>type t_pbrk.c
#include <stdio.h>

extern          char *strpbrk();

main()
{
    puts(strpbrk("第1水準漢字","2 3 4"));……第1引数のなかに，第2引数のどれか
}                                             1文字がないかを調べる．この場合，
                                              NULLが返されることを期待している
C>t_pbrk
1水準漢字
└──── "1"の第1バイトと"2"の第1バイトが一致してしまった
```

図3-21　strpbrk関数に全角文字列を渡した場合

このような関数には，MS-Cでいうと，strchr，strrchr，strpbrk，strspn，strtokなどがあります．これらの関数は，引数に前処理を施すなどの方法では対策をとることができません．代わりの関数を新たに作成する必要があります．

MS-Cでは，これらの関数の代わりに日本語処理関数のjstrchr，jstrrchr，jstrmatch，jstrskip，jstrtokを使用します．なお，strcspnについては，jstrmatchを利用して同等な関数を作成することが可能です．

文字列の変換を行う関数

シフトJISでは，全角文字の第2バイトと半角文字の分布が重なっているために，MS-Cのstrlwrやstruprなどのような引数の文字列を変換する関数に全角文字列を渡すと，変換してほしくない全角

文字まで変換されてしまう場合があります(図3-22)．また，処理系によっては，変換対象となる文字かどうかの判断に際し，8ビット目をマスクしてから比較するようなものもあり，その場合，EUCを使用していても異常な結果を招きます．

```
C>type t_lwr.c
#include <stdio.h>

extern          char *strlwr();

main()
{
    puts(strlwr("ＡＢＣ０１２"));
}

C>t_lwr
ＡＢＣＰＱＲ
C>
```

"０１２"の第２バイトが半角の英大文字に相当するため，変換されてしまった

図3-22　全角文字列に対して strlwr 関数をかける

この場合，MS-C では jtolower や jtoupper などの関数を利用して1文字ずつ変換していくようにすれば解決します(全角文字のアルファベットが対象)．半角のアルファベットだけを変換したい場合は，全角文字かどうかの判断を行ってから変換するという手順を踏みます(リスト3-8)．

```
 1: #include <ctype.h>
 2:
 3: #define _iskanji(c) (((c) & 0xff)>=0x81 && ((c) & 0xff)<=0x9f || ¥
 4:                      ((c) & 0xff)>=0xe0 && ((c) & 0xff)<=0xfc)
 5:
 6: char *htolower(char *s)
 7: {
 8:     char *os = s;
 9:
10:     while (*s != '¥0'){
11:     if (_iskanji(*s)){
12:         s += 2;
13:         continue;
14:     }
15:     if (*s >= 'A' && *s <= 'Z')
16:         *s += ('a'-'A');
17:     ++s;
18:     }
19:     return os;
20: }
```

全角文字は変換しない（11〜14行）

リスト3-8　HTOLOWER.C

文字列の切断を行う関数

文字列のnバイト目までコピーするといった関数に，nバイト目が全角文字の第1バイトであるような文字列を渡すと，全角文字の第1バイトのみの中途半端な文字が生まれてしまいます．このような文字を画面に出力すると，画面表示の異常を引き起こす場合があります．

このような事故を防ぐために，nバイト目が全角文字の第1バイトであれば，「n−1バイト」だけコピーするとか，「n文字」コピーするといった指定をする関数が必要になります．MS-Cのjstrncpy，jstrncatでは「n文字」の仕様になっています．

■ 独自の内部処理コードの作成

ここまで，OSが使用している内部処理用漢字コードを使うことを前提に話を進めてきましたが，理想を述べると，全角文字1文字も1単位として扱えるlong charのような型で全角／半角を問わずに統一的に扱えるようになると，処理が大幅にやりやすくなります．しかし，1文字1単位を実現するのに，short int型の配列に文字を格納する方法をとると，**図3-23**に示すように文字列編集のライブラリ関数はおろか，入出力関係の関数も満足に利用できなくなってしまいます．しかし，この方法は，入出力の際に変換を行うことにすれば，比較的魅力のある方法といえるでしょう．

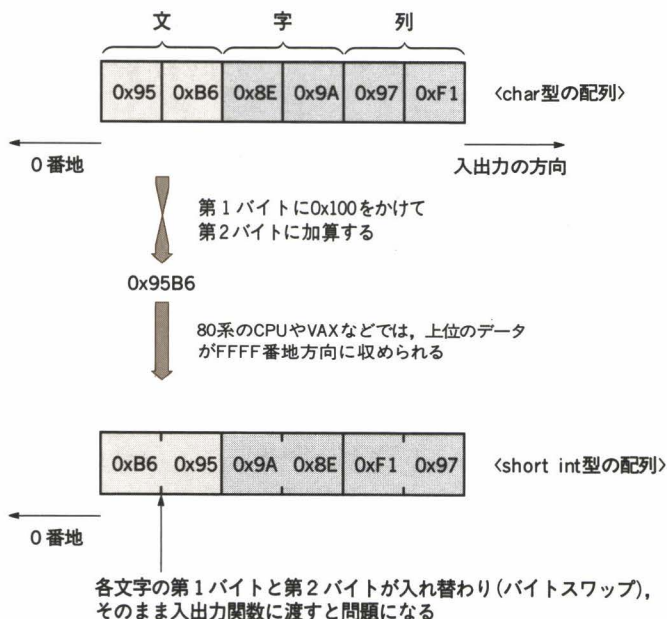



図3-23　80系CPU，VAXにおけるバイトスワップ

日本語処理を行う場合，ライブラリ関数は使いものにならなくてもともとですから，いくつか使える関数が存在していればもうけものです．そこで，内部コードへの変換をする際に，半角文字を表現するのに半角文字コードを単に char 型から short int 型に拡張するだけでなく，0x2000 などの値を加算するようにします(**リスト 3-9**)．そうすることで，内部コードの配列を char 型の配列としたときもヌル文字が現れずにすみ，strcpy などの基本的なライブラリ関数ぐらいは使用できるようになるので便利です．

```
 1: #include <stdio.h>
 2:
 3: #define ANK_S    (' ' << 8)
 4: #define ESC      '¥033'
 5:
 6: #define KIO_LEN 3
 7:
 8: #define iski(p)  (((p)[0]==ki_s[0]  && (p)[1]==ki_s[1]  && ¥
 9:                    (p)[2]==ki_s[2]) || ¥
10:                   ((p)[0]==ki_s2[0] && (p)[1]==ki_s2[1] && ¥
11:                    (p)[2]==ki_s2[2]))
12:
13: #define isko(p)  (((p)[0]==ko_s[0]  && (p)[1]==ko_s[1]  && ¥
14:                    (p)[2]==ko_s[2]) ||Y
15:                   ((p)[0]==ko_s2[0] && (p)[1]==ko_s2[1] && ¥
16:                    (p)[2]==ko_s2[2]) ||¥
17:                   ((p)[0]==ko_s3[0] && (p)[1]==ko_s3[1] && ¥
18:                    (p)[2]==ko_s3[2]))
19:
20: #define eoss(c)          (((c) & 0xff) == 0 || ((c) & 0xff00) == 0)
21:
22: #define _isank(c)        (((c) & 0xff00) == ANK_S)
23:
24: #define add_shift(c, s)  ((c)[0]=(s)[0], (c)[1]=(s)[1], (c)[2]=(s)[2])
25:
26: unsigned char ki_s[]  = "¥033$B";
27: unsigned char ki_s2[] = "¥033$@";
28: unsigned char ko_s[]  = "¥033(J";
29: unsigned char ko_s2[] = "¥033(B";
30: unsigned char ko_s3[] = "¥033(H";
31:
32: crunch(unsigned short int *ss, unsigned char *cs)
33: {
34:     int len;
35:     int kanji;
36:
37:     len = 0;
38:     kanji = 0;
39:     while (*cs != '¥0'){
40:         if (*cs == ESC){
41:             if (kanji){
42:                 if (isko(cs)){
```

（3行目の注）半角文字の識別用文字（short int型の配列に半角文字を入れるときに加算する）

（20行目の注）内部コード文字列の終端の判別をするマクロ

（22行目の注）半角文字を表すコードかどうかを判別するマクロ

```
43:                     kanji = 0;
44:                     cs += KIO_LEN;
45:                     continue;
46:                 }
47:             } else {
48:                 if (iski(cs)){
49:                     kanji = -1;
50:                     cs += KIO_LEN;
51:                     continue;
52:                 }
53:             }
54:         }
55:         if (kanji){
56:             *ss = *cs << 8 | *(cs+1);
57:             ++len;
58:             ++ss;
59:             cs += 2;
60:         } else {
61:             *ss = ANK_S | *cs;
62:             ++len;
63:             ++ss;
64:             ++cs;
65:         }
66:     }
67:     *ss = 0;
68:     return len;
69: }
70:
71: decrunch(unsigned char *cs, unsigned short int *ss)
72: {
73:     int len;
74:     int kanji;
75:
76:     len = 0;
77:     kanji = 0;
78:     while (!eoss(*ss)){
79:         if (_isank(*ss)){
80:             if (kanji){
81:                 kanji = 0;
82:                 add_shift(cs, ko_s);
83:                 len += KIO_LEN;
84:                 cs += KIO_LEN;
85:             }
86:             *cs = *ss & 0xff;
87:             ++len;
88:             ++ss;
89:             ++cs;
90:         } else {
91:             if (!kanji){
92:                 kanji = -1;
93:                 add_shift(cs, ki_s);
94:                 len += KIO_LEN;
95:                 cs += KIO_LEN;
96:             }
```

（43〜53行）シフトコードの除去と漢字モードの設定

（56行）第1バイトと第2バイトを合わせている

（80〜85行）KOコードの付加

（91〜96行）KIコードの付加

```
 97:                *cs = (*ss & 0xff00) >> 8;    } 第1バイトと第2バイトに分ける
 98:                *(cs+1) = *ss & 0xff;         }
 99:                len += 2;
100:                ++ss;
101:                cs += 2;
102:            }
103:        }
104:        if (kanji){
105:            add_shift(cs, ko_s);
106:            len += KIO_LEN;
107:            cs += KIO_LEN;
108:        }
109:        *cs = '¥0';
110:        return len;
111: }
```

リスト 3-9　CRUNCH.C

```
 1: #include <stdio.h>
 2: 
 3: main(int argc, char *argv[])
 4: {
 5:     char  c2[256];
 6:     short int i[256];……………………内部コードの格納領域
 7:     int   len, ii, ti;
 8: 
 9:     len = crunch(i, argv[1]);……………………内部コードへの変換
10:     for (ii = 0; ii < len-4; ii += 3){
11:         ti = i[ii]; i[ii] = i[ii+1];                }
12:         i[ii+1] = i[ii+2]; i[ii+2] = ti;            } 全角／半角の判断をしなくても編集できる
13:     }                                               }
14:     decrunch(c2, i);
15:     puts(c2);
16: }
```

リスト 3-10　ICTEST.C

文字列の解析を行う関数や変換を行う関数については，すべて自分で面倒をみてやる必要があります．また，入出力を頻繁に行うプログラムでは，文字コードの変換によるオーバーヘッドが問題になります．さらに，文字列中の文字がどの位置に表示されるのか，文字列のスキャンを行わなければ知ることができないなど，いくつかの問題点をかかえています．それでも，JIS コードや OS の内部処理用コードをそのまま使うよりは，はるかに楽に処理できます．とくに，全角文字と半角文字の判断が即座にできる点や，第1バイトと第2バイトの判別をしなくてもすむようになりますから，文字列の解析を行うようなプログラムでは開発効率や実行速度が大きく向上するでしょう．この方法は，将来 long char 型がサポートされるようになった場合にも応用が可能な手法ですから，いまからこのようなプログラムを組んでおくのもよいかもしれません．

3.5 サンプルプログラム —JFOLD—

この章のまとめとして，3.3節でも一部を紹介したサンプルプログラム「JFOLD」(シフトJIS版／JIS版／EUC版)を紹介します．

JFOLDコマンドは長い行を一定の幅で折りたたむコマンドです．普通，エディタは1行の長さに制限がありますが(たとえばMS-DOSのEDLINコマンドでは255文字)，ワープロでは1行の長さに制限はありません．そこでワープロの出力をエディタで編集する際に，問題なく編集できるようにテキストの変換を行う，つまり適当な長さで改行コードを挿入するのがJFOLDコマンドです．また日本語文章を扱うことを考慮して，行頭禁則処理機能と全角文字のあいだに改行コードがはいり込むのを防ぐ機能を加えてあります．

行頭禁則処理とは，行の先頭に「，」や「．」などの文字が現れないようにする処理をいいます．これとは逆に，行の終わりに「『」(かっこの開き)などが現れないようにする処理を行末禁則処理といいます．

ここでは3種類のJFOLDコマンドを紹介します．シフトJISコード，JISコードそれにEUCコードに対応したもので，どれも行頭禁則処理機能を組み込んであり，使い方はまったく同じです．ただし行末禁則処理については考慮していませんから，利用に際しては注意してください．

JFOLDコマンドのオプションを**表3-3**に示します．

オプション	機　　能
−a	全角文字の処理機能をなくす
−<数字>	横幅を指定する．デフォルトは72文字(半角)
−k<禁則文字列>	行頭禁則を行う文字列を指定する

表3-3　JFOLDコマンドのオプション一覧

JFOLDコマンドは，たとえば次のように起動すると，textというファイルを80文字の横幅で「，．、。ー」について行頭禁則を行い，変換した結果を標準出力に出力します．

```
A>jfold  -80  -k,.、。ー  text
```

また普通，禁則文字はそれほど頻繁に変更されるものではないので，環境変数に〝KINSOKU〟を設け，この変数が設定されている場合には，そこで設定されている文字列を行頭禁則文字としてプログラムを実行します(-aが指定された場合は無視される)．

```
A>type bunshou
  全角文字の問題ではなく、日本語の文章の問題として、行末の句読点や括弧の問題があ
ります。
  表示幅などの関係で句読点の直前の文字で改行されると、行頭に句読点が来ることにな
ってしまいます。
  このプログラムは、単に行を切り刻むだけでなく、同時に禁則処理を行うことで、正し
い日本語になるように処理します。

A>jfold -52 -k。、 bunshou
  全角文字の問題ではなく、日本語の文章の問題として、
行末の句読点や括弧の問題があります。
  表示幅などの関係で句読点の直前の文字で改行されると、
行頭に句読点が来ることになってしまいます。
  このプログラムは、単に行を切り刻むだけでなく、同時
に禁則処理を行うことで、正しい日本語になるように処理
します。

A>
```

……禁則処理のため53行目に表示される

図3-24 JFOLDコマンドの実行結果

プログラムとしてはとくに複雑な点はないと思いますが，禁則の判断の部分が少々理解しづらいかもしれません．全角文字のときには2バイトで1文字とする処理が必要なので，その点を明確にするために，全角文字かどうかの判断をプログラムのメインループのはじめで行っています．

またJISコード版では，半角文字から全角文字へシフトコードを「ESC」,「$」,「B」の3バイト，全角文字から半角文字へのシフトコードを「ESC」,「(」,「J」としてあります．このプログラムでは入力文字列と出力文字列のそれぞれについて，全角文字か半角文字かを判断するフラグを用意し，それぞれを独立に管理しています．この方法をとることにより，入力文字列についてKIとKOが連続してテキストにはいっている場合の余分なシフトコードを削除できます．

```
 1: /*
 2:  *  jfold - fold long lines with shift-JIS KANJI
 3:  *
 4:  *    Written by Masato Minda
 5:  *
 6:  *    Copyright (C) 1987 by Masato Minda
 7:  *
 8:  *    modified Oct. 1,1990 by Katsu-F
 9:  */
10: static char sccsid[] = "@(#)jfold.c      1.1 (MinMin)  87/05/26";
11: static char v_upid[] = "@(#)             1.2 (Katsu-F) 90/10/01";
12:
13: #include  <stdio.h>
14: #include  <stdlib.h>
15: #include  <ctype.h>
16: #include  <io.h>
```

```
17:
18: #define  ARGV0          /*  argv[0] is valid  */ ……………argv[0]が有効な
19: #define  TRUE   1                                        OS(UNIXなど)では
20: #define  FALSE  0                                        プログラム名に利
21:                                                          用する
22: #define  LEN     72     /*  default line length  */ ……デフォルトの改行幅
23: #define  KINMAX  20     /*  kinsoku moji table size  */
24: #define  TABSIZ   8     /*  default tab-size  */
25:
26: #ifdef  iskanji
27: #   undef  iskanji
28: #endif
29:                         iskanjiのマクロで利用するテーブルを作るためのマクロ………
30: #define  _iskanji(c)    ((c)>=0x81&&(c)<=0x9f||(c)>=0xe0&&(c)<=0xfc)…
31: #define  iskanji(c)     (_ktype[(c) & 0xff])…………シフトJISの全角文字の第1
32:                                                  バイトをチェックするマクロ
33: char    *Prgnam = "jfold";
34: char     _ktype[0x100];
35: int      Len = LEN;
36: int      Aflag = FALSE;
37: int      Kflag = FALSE;
38: int      Kcount = 0;
39: int      Kinsoku[KINMAX];
40:
41: void     iniktype (void)
42: {
43:     register int     i;
44:
45:     for (i = 0; i < 0x100; i++) {
46:         if (_iskanji (i)) {
47:             _ktype[i] = 1;               iskanjiのマクロで利用するテーブルを作る
48:         } else {
49:             _ktype[i] = 0;
50:         }
51:     }
52: }
53:
54: void     uniniktp (void)
55: {
56:     register int     i;
57:                                          iskanjiが常に偽となるようにテーブルを
58:     for (i = 0; i < 0x100; i++) {        リセットする(-aオプション時)
59:         _ktype[i] = 0;
60:     }
61: }
62:
63: char     *addkin (char *p)………………禁則文字をテーブルに登録する関数
64: {
65:     while (*p) {
66:         if (iskanji (*p)) {
67:             Kinsoku[Kcount] = (*p & 0xff) << 8 | (*(p + 1) & 0xff);
68:             p += 2;
69:         } else {
70:             Kinsoku[Kcount] = *p & 0xff;
71:             ++p;
72:         }
```

```
 73:         if (++Kcount > KINMAX) {
 74:             fputs (Prgnam, stderr);
 75:             fputs (": too many kinsoku charactor¥n", stderr);
 76:             exit (1);
 77:         }
 78:     }
 79:     return p;
 80: }
 81:
 82: int     kintbl (int c1, int c2)
 83: {
 84:     register int    c0, i;
 85:
 86:     c0 = ((c1 & 0xff) << 8) | (c2 & 0xff);
 87:     for (i = 0; i < Kcount; i++) {
 88:         if (c0 == Kinsoku[i]) {
 89:             return 1;
 90:         }
 91:     }
 92:     return 0;
 93: }
 94:
 95: void    usage (void)
 96: {
 97:     fputs ("Usage: ", stderr);
 98:     fputs (Prgnam, stderr);
 99:     fputs (" [-a] [-k<kinsoku>] [-<n>] [ <file> ... ]¥n", stderr);
100:     exit (1);
101: }
102:
103: void    fold (FILE *fp)
104: {
105:     register int    col, c1, c2;
106:
107:     col = 0;
108:     while ((c1 = getc (fp)) != EOF) {
109:         if (c1 == '¥n') {       /*  ? code is newline   */
110:             putchar (c1);       /*  new line  */
111:             col = 0;
112:             continue;
113:         }
114:         if (iskanji (c1)) {     /*  ? kanji   */
115:             if ((c2 = getc (fp)) == EOF) {      /*  get next char   */
116:                 putchar (c1);       /*  can't execute this line   */
117:                 return;
118:             }
119:             if (col >= Len - 1) {       /*  ? col is limit   */
120:                 if (Kflag && kintbl (c1, c2)) {     /*  ? kinsoku   */
121:                     putchar (c1);
122:                     putchar (c2);
123:                     col += 2;
124:                     continue;
125:                 }
126:                 putchar ('¥n');
127:                 col = 0;
128:             }
```

- 74〜76行: fprintf関数を利用していないのは，オブジェクトのサイズを小さくするため
- 82行: C1，C2からなる文字が，禁則文字であるかどうかをチェックする関数
- 103行: メインルーチン
- 105行 col: 改行幅を管理する変数
- 109〜110行: まず改行コードをチェックする
- 115行: 1バイト目が漢字コードの第1バイトであるとき，漢字コードの第2バイトを読み込む
- 116〜117行: 正しいテキストファイルを入力したときは，この行は実行されない
- 119〜124行: 禁則の処理
- 126〜127行: 禁則文字でないときは，改行コードを出力して改行

```
129:                 putchar (c1);     } 文字の出力
130:                 putchar (c2);     }
131:                 col += 2;
132:            } else { ………………漢字でない場合(基本的には漢字の場合と処理は同じ)
133:                 if (col >= Len) {      /*  ? col is limit  */
134:                      if (Kflag && kintbl (0, c1)) {      /*  ? kinsoku  */
135:                           putchar (c1);
136:                           col += 1;
137:                           continue;
138:                      }
139:                      putchar ('¥n');
140:                      col = 0;
141:                 }
142:                 if (c1 == '¥t') {      /*  ? tab code  */
143:                      if ((col = (col / TABSIZ + 1) * TABSIZ - 1) < Len) {
144:                           ++col;
145:                           putchar (c1);
146: TABコードの処理…      continue;
147:                      }
148:                      putchar ('¥n');
149:                      putchar (c1);
150:                      col = TABSIZ;
151:                      continue;
152:                 }
153:                 putchar (c1);
154:                 col += 1;
155:            }
156:       }
157: }
158:
159: void    main (int argc, char **argv)
160: {
161:      FILE   *fp;
162:      char   *p;
163:      char   *getenv ();
164:
165: #ifdef  ARGV0
166:      Prgnam = *argv;
167: #endif
168:      iniktype ();
169:      while (--argc && **++argv == '-') {
170:           while (*++*argv) {
171:                if (isdigit (**argv)) {
172:                     if ((Len = atoi (*argv)) == 0) {
173:                          usage ();
174:                     }
175:                     while (isdigit (*(*argv + 1))) {
176:                          ++*argv;
177:                     }
178:                     continue;
179:                }
180:                switch (**argv) {
181:                case 'a':                                  オプションおよび，
182:                     Aflag = TRUE;                         改行幅の処理
183:                     continue;
```

```
184:                case 'k':
185:                    Kflag = TRUE;
186:                    *argv = addkin (*argv + 1) - 1;
187:                    break;
188:                default:
189:                    usage ();
190:                }
191:            }
192:        }
193:        ++argc;
194:        --argv;
195:        if ((p = getenv ("KINSOKU")) != NULL) {
196:            Kflag = TRUE;
197:            addkin (p);
198:        }
199:        if (Aflag) {
200:            Kflag = FALSE;
201:            uniniktp ();
202:        }
203:        if (argc == 1) {
204:            if (isatty (fileno (stdin))) {
205:                usage ();
206:            }
207:            fold (stdin);
208:            exit (0);
209:        }
210:        while (--argc) {
211:            if ((fp = fopen (*++argv, "r")) == NULL) {
212:                fputs (Prgnam, stderr);
213:                fputs (": can't open ", stderr);
214:                fputs (*argv, stderr);
215:                fputs ("¥n", stderr);
216:                exit (1);
217:            }
218:            fold (fp);
219:            fclose (fp);
220:        }
221:        exit (0);
222: }
```

環境変数の"KINSOKU"をチェックし，存在すれば禁則処理を行うためのセットアップを行う

-aオプションが指定されたときは，漢字の処理を禁止する

標準入力がttyならば，Usageメッセージを出力する

リスト 3-11 JFOLD.C(シフトJIS版)

```
 1: /*
 2:  *  jfold - fold long lines with JIS KANJI
 3:  *
 4:  *    Written by Masato Minda
 5:  *
 6:  *    Copyright (C) 1987 by Masato Minda
 7:  *
 8:  *    Modified Oct. 1, 1990 by Katsu-F
 9:  */
10: static char sccsid[] = "@(#)jfold2.c     1.1 (MinMin)   87/05/26";
11: static char v_upid[] = "@(#)             1.2 (Katsu-F) 90/10/01";
```

```
12:
13: #include  <stdio.h>
14: #include  <stdlib.h>
15: #include  <ctype.h>
16: #include  <io.h>
17: #include  <string.h>
18:
19: #define  ARGV0            /*  argv[0] is valid  */
20: #define  TRUE   1
21: #define  FALSE  0
22:
23: #define  LEN      72      /*  default line length  */
24: #define  KINMAX  20       /*  kinsoku moji table size  */
25: #define  TABSIZ   8       /*  default tab-size  */
26:
27: #define  KANJI    1       /*  mode is kanji  */
28: #define  ANKCH    0       /*  mode is ank  */
29: #define  ESC    '¥033'    /*  Escape code  */
30: #define  KI     "$B"      /*  kanji shift-in  */
31: #define  KO     "(J"      /*  kanji shift-out  */
32:
33: char   *Prgnam = "jfold";
34: int     Len = LEN;
35: int     Aflag = FALSE;
36: int     Kflag = FALSE;
37: int     Kcount = 0;
38: int     Kinsoku[KINMAX];
39: char   *Ki = KI;………漢字イン
40: char   *Ko = KO;………漢字アウト
41: int     Lki;………………漢字インの文字列の長さ
42: int     Lko;………………漢字アウトの文字列の長さ
43:
44: char    *addkin (char *p)…………シフトJISコード用JFOLDのaddkin関数と同様の処理
45: {
46:     register int     mode;………着目している文字が全角文字か半角文字かを記憶
47:
48:     mode = ANKCH;
49:     while (*p) {
50:         if (*p == ESC) {
51:             ++p;
52:             if (!strncmp (p, Ki, Lki)) {
53:                 mode = KANJI;
54:                 p += Lki;
55:                 continue;
56:             }
57:             if (!strncmp (p, Ko, Lko)) {      KI，KOをチェックして
58:                 mode = ANKCH;                 モードを設定する
59:                 p += Lko;
60:                 continue;
61:             }
62:         }
63:         if (mode == KANJI) {
64:             Kinsoku[Kcount] = (*p & 0xff) << 8 | (*(p + 1) & 0xff);
65:             p += 2;
66:         } else {
67:             Kinsoku[Kcount] = *p & 0xff;
68:             ++p;
69:         }
```

```
 70:         if (++Kcount > KINMAX) {
 71:             fputs (Prgnam, stderr);
 72:             fputs (": too many kinsoku charactor.¥n", stderr);
 73:             exit (1);
 74:         }
 75:     }
 76:     return p;
 77: }
 78:
 79: int     kintbl (int c1, int c2)
 80: {
 81:     register int     c0, i;
 82:
 83:     c0 = ((c1 & 0xff) << 8) | (c2 & 0xff);
 84:     for (i = 0; i < Kcount; i++) {
 85:         if (c0 == Kinsoku[i]) {
 86:             return 1;
 87:         }
 88:     }
 89:     return 0;
 90: }
 91:
 92: void    usage (void)
 93: {
 94:     fputs ("Usage: ", stderr);
 95:     fputs (Prgnam, stderr);
 96:     fputs (" [-a] [-k<kinsoku>] [-<n>] [ <file> ... ]¥n", stderr);
 97:     exit (1);
 98: }
 99:
100: void    unknown (void)
101: {
102:     fputs (Prgnam, stderr);
103:     fputs (": unknown escape sequence¥n", stderr);
104: }
105:
106: void    fold (FILE *fp)
107: {                                ┌─→入力文字列が漢字かどうかを管理する変数
108:     register int     imode, omode, col, c1, c2;
109:     register char    *p;         └─→出力文字列が漢字かどうかを管理する変数
110:
111:     omode = imode = ANKCH;
112:     col = 0;
113:     while ((c1 = getc (fp)) != EOF) {
114:         if (c1 == '¥n') {      /*  ? code is newline  */
115:             if (omode == KANJI) {
116:                 putchar (ESC);          出力文字のモードが全角文字ならば,
117:                 fputs (Ko, stdout);     半角文字のモードに直す
118:             }
119:             putchar (c1);      /*  new line  */
120:             col = 0;
121:             omode = ANKCH;
122:             continue;
123:         }
124:         if (c1 == ESC) {       /*  ? ESC  */
125:             if ((c1 = getc (fp)) == *Ki) {
126:                 for (p = Ki + 1; *p; p++) {
127:                     if ((c1 = getc (fp)) != *p)  {
```

```
128:                    unknown ();
129:                    goto next;
130:                }
131:            }
132:            imode = KANJI;
133:        } else if (c1 == *Ko) {
134:            for (p = Ko + 1; *p; p++) {
135:                if ((c1 = getc (fp)) != *p)  {
136:                    unknown ();
137:                    goto next;
138:                }
139:            }
140:            imode = ANKCH;
141:        } else {
142:            unknown ();
143:        }
144:        continue;
145:    }
146:    if (imode == KANJI) {    /*  ? kanji mode  */
147:        if (omode == ANKCH) {
148:            putchar (ESC);
149:            fputs (Ki, stdout);
150:            omode = KANJI;
151:        }
152:        if ((c2 = getc (fp)) == EOF) {     /*  get next char  */
153:            putchar (c1);    /*  can't execute this line  */
154:            return;
155:        }
156:        if (col >= Len - 1) {    /*  ? col is limit  */
157:            if (Kflag && kintbl (c1, c2)) {     /*  ? kinsoku  */
158:                putchar (c1);
159:                putchar (c2);
160:                col += 2;
161:                continue;
162:            }
163:            putchar (ESC);
164:            fputs (Ko, stdout);
165:            putchar ('¥n');
166:            putchar (ESC);
167:            fputs (Ki, stdout);
168:            col = 0;
169:        }
170:        putchar (c1);
171:        putchar (c2);
172:        col += 2;
173:    } else {
174:        if (omode == KANJI) {
175:            putchar (ESC);
176:            fputs (Ko, stdout);
177:            omode = ANKCH;
178:        }
179:        if (col >= Len) {    /*  ? col is limit  */
180:            if (Kflag && kintbl (0, c1)) {     /*  ? kinsoku  */
181:                putchar (c1);
182:                col += 1;
183:                continue;
184:            }
185:            putchar ('¥n');
186:            col = 0;
```

注釈:
- 128〜143行：エスケープシーケンスにしたがって入力文字列が全角文字か半角文字かを設定する
- 146行：入力文字列が漢字のとき
- 147〜150行：出力文字のモードが半角文字なら，全角文字のモードに直す
- 152行：次の1文字を入力よりひろう
- 156行：改行幅に達したか？
- 157〜162行：禁則文字の処理
- 163〜169行：改行コマンドを出力
- 174〜177行：出力文字のモードが全角文字のモードなら半角文字のモードに直す

```
187:             }
188:             if (c1 == '¥t') {    /*  ? tab code  */
189:                 if ((col = (col / TABSIZ + 1) * TABSIZ - 1) < Len) {
190:                     ++col;
191:                     putchar (c1);
192:                     continue;
193:                 }
194:                 putchar ('¥n');
195:                 putchar (c1);
196:                 col = TABSIZ;
197:                 continue;
198:             }
199:             putchar (c1);
200:             col += 1;
201:         }
202:     next:
203:         ;
204:     }
205:     if (omode == KANJI) {
206:         putchar (ESC);
207:         fputs (Ko, stdout);
208:     }
```

最終的に出力モードが全角文字ならば，半角文字のモードに直す

```
209: }
210:
211: void    main (int argc, char **argv)
212: {
213:     FILE   *fp;
214:     char   *p;
215:     char   *getenv ();
216:
217: #ifdef  ARGV0
218:     Prgnam = *argv;
219: #endif
220:     Lki = strlen (Ki);
221:     Lko = strlen (Ko);
222:     while (--argc && **++argv == '-') {
223:         while (*++*argv) {
224:             if (isdigit (**argv)) {
225:                 if ((Len = atoi (*argv)) == 0) {
226:                     usage ();
227:                 }
228:                 while (isdigit (*(*argv + 1))) {
229:                     ++*argv;
230:                 }
231:                 continue;
232:             }
233:             switch (**argv) {
234:             case 'a':
235:                 Aflag = TRUE;
236:                 continue;
237:             case 'k':
238:                 Kflag = TRUE;
239:                 *argv = addkin (*argv + 1) - 1;
240:                 break;
241:             default:
242:                 usage ();
243:             }
244:         }
245:     }
```

```
246:      ++argc;
247:      --argv;
248:      if ((p = getenv ("KINSOKU")) != NULL) {
249:          Kflag = TRUE;
250:          addkin (p);
251:      }
252:      if (Aflag) {
253:          Kflag = FALSE;
254:      }
255:      if (argc == 1) {
256:          if (isatty (fileno (stdin))) {
257:              usage ();
258:          }
259:          fold (stdin);
260:          exit (0);
261:      }
262:      while (--argc) {
263:          if ((fp = fopen (*++argv, "r")) == NULL) {
264:              fputs (Prgnam, stderr);
265:              fputs (": can't open ", stderr);
266:              fputs (*argv, stderr);
267:              fputs ("¥n", stderr);
268:              exit (1);
269:          }
270:          fold (fp);
271:          fclose (fp);
272:      }
273:      exit (0);
274: }
```

リスト 3-12　JFOLD2.C(JIS 版)

```
 1: /*
 2:  *  jfold - fold long lines with EUC KANJI
 3:  *
 4:  *     Written by Masato Minda
 5:  *
 6:  *     Copyright (C) 1987 by Masato Minda
 7:  *
 8:  *     Modified Oct. 1, 1990 by Katsu-F
 9:  */
10: static char sccsid[] = "@(#)jfold3.c     1.1 (MinMin)   87/05/26";
11: static char v_upid[] = "@(#)             1.2 (Katsu-F)  90/10/01";
12:
13: #include  <stdio.h>
14: #include  <stdlib.h>
15: #include  <ctype.h>
16: #include  <io.h>
17:
18: #define  ARGV0           /*  argv[0] is valid  */
19: #define  TRUE   1
20: #define  FALSE  0
21:
```

argv[0]が有効なOS(UNIXなど)ではプログラム名に利用する

```
22: #define  LEN      72      /*  default line length   */………デフォルトの改行幅
23: #define  KINMAX   20      /*  kinsoku moji table size   */
24: #define  TABSIZ    8      /*  default tab-size  */
25:
26: #ifdef  iskanji
27: #   undef  iskanji
28: #endif
29: #ifdef  iskana                        iskanjiのマクロで利用するテーブルを作るためのマクロ
30: #   undef  iskana
31: #endif
32:
33: #define  _iskanji(c)      ((c)>=0xa1&&(c)<=0xfe)……iskanji, iskanaは独自に定義するので，もしすでに定義されている場合は，未定義にする
34: #define  iskanji(c)       (_ktype[(c) & 0xff])……EUCの全角文字をチェックするマクロ
35: #define  iskana(c)        ((c)==0x8e) ……………EUCの半角カタカナ文字シフトコードをチェックするマクロ
36:
37: char    *Prgnam = "jfold";
38: char     _ktype[0x100];
39: int      Len = LEN;
40: int      Aflag = FALSE;
41: int      Kflag = FALSE;
42: int      Kcount = 0;
43: int      Kinsoku[KINMAX];
44:
45: void     iniktype (void)
46: {
47:     register int      i;
48:
49:     for (i = 0; i < 0x100; i++) {
50:         if (_iskanji (i)) {            iskanjiで使うテーブルのセットアップ
51:             _ktype[i] = 1;
52:         } else {
53:             _ktype[i] = 0;
54:         }
55:     }
56: }
57:
58: void     uniniktp (void)
59: {
60:     register int     i;              iskanjiが常に偽になるように
61:                                      テーブルをクリアする
62:     for (i = 0; i < 0x100; i++) {
63:         _ktype[i] = 0;
64:     }
65: }
66:
67: char     *addkin (char *p)…………………禁則文字をテーブルに登録する関数
68: {
69:     while (*p) {
70:         if (iskanji (*p) || iskana (*p)) {
71:             Kinsoku[Kcount] = (*p & 0xff) << 8 | (*(p + 1) & 0xff);
72:             p += 2;
73:         } else {
74:             Kinsoku[Kcount] = *p & 0xff;
75:             ++p;
76:         }
```

```
 77:         if (++Kcount > KINMAX) {
 78:             fputs (Prgnam, stderr);
 79:             fputs (": too many kinsoku charactor¥n", stderr);
 80:             exit (1);
 81:         }
 82:     }
 83:     return p;
 84: }
 85:
 86: int     kintbl (int c1, int c2)
 87: {
 88:     register int    c0, i;
 89:
 90:     c0 = ((c1 & 0xff) << 8) | (c2 & 0xff);
 91:     for (i = 0; i < Kcount; i++) {
 92:         if (c0 == Kinsoku[i]) {
 93:             return 1;
 94:         }
 95:     }
 96:     return 0;
 97: }
 98:
 99: void    usage (void)
100: {
101:     fputs ("Usage: ", stderr);
102:     fputs (Prgnam, stderr);
103:     fputs (" [-a] [-k<kinsoku>] [-<n>] [ <file> ... ]¥n", stderr);
104:     exit (1);
105: }
106:
107: void    fold (FILE *fp)
108: {
109:     register int    col, c1, c2;
110:
111:     col = 0;
112:     while ((c1 = getc (fp)) != EOF) {
113:         if (c1 == '¥n') {       /*  ? code is newline  */
114:             putchar (c1);       /*  new line  */
115:             col = 0;
116:             continue;
117:         }
118:         if (iskanji (c1)) {    /*  ? kanji  */
119:             if ((c2 = getc (fp)) == EOF) {     /*  get next char  */
120:                 putchar (c1);     /*  can't execute this line  */
121:                 return;
122:             }
123:             if (col >= Len - 1) {     /*  ? col is limit  */
124:                 if (Kflag && kintbl (c1, c2)) {     /*  ? kinsoku  */
125:                     putchar (c1);
126:                     putchar (c2);
127:                     col += 2;
128:                     continue;
129:                 }
130:                 putchar ('¥n');
131:                 col = 0;
132:             }
```

（79行目）fprintf関数を利用していないのは，オブジェクトのサイズを小さくするため

（86〜97行目）c1(上位バイト),c2(下位バイト)で与えられる文字が，禁則文字かどうかをチェックする関数

（107行目）メインルーチン

（109行目 col）現在の桁位置を示す変数

（113〜114行目）まず改行コードをチェックする

（123〜128行目）禁則文字の場合は改行せずに，そのままその文字を出力する

```
133:             putchar (c1);
134:             putchar (c2);
135:             col += 2;
136:         } else if (iskana (c1)) {    /* ? kana  */
137:             if ((c2 = getc (fp)) == EOF) {      /*  get next char  */
138:                 putchar (c1);     /*  can't execute this line  */
139:                 return;
140:             }
141:             if (col >= Len) {     /*  ? col is limit   */
142:                 if (Kflag && kintbl (c1, c2)) {      /*  ? kinsoku  */
143:                     putchar (c1);
144:                     putchar (c2);
145:                     col += 1;
146:                     continue;
147:                 }
148:                 putchar ('¥n');
149:                 col = 0;
150:             }
151:             putchar (c1);
152:             putchar (c2);
153:             ++col;
154:         } else {
155:             if (col >= Len) {     /*  ? col is limit   */
156:                 if (Kflag && kintbl (0, c1)) {      /*  ? kinsoku  */
157:                     putchar (c1);
158:                     col += 1;
159:                     continue;
160:                 }
161:                 putchar ('¥n');
162:                 col = 0;
163:             }
164:             if (c1 == '¥t') {     /*  ? tab code   */
165:                 if ((col = (col / TABSIZ + 1) * TABSIZ - 1) < Len) {
166:                     ++col;
167:                     putchar (c1);
168:                     continue;
169:                 }
170:                 putchar ('¥n');
171:                 putchar (c1);
172:                 col = TABSIZ;
173:                 continue;
174:             }
175:             putchar (c1);
176:             col += 1;
177:         }
178:     }
179: }
180:
181: void    main (int argc, char **argv)
182: {
183:     FILE    *fp;
184:     char    *p;
185:     char    *getenv ();
186:
187: #ifdef  ARGV0
188:     Prgnam = *argv;
189: #endif
```

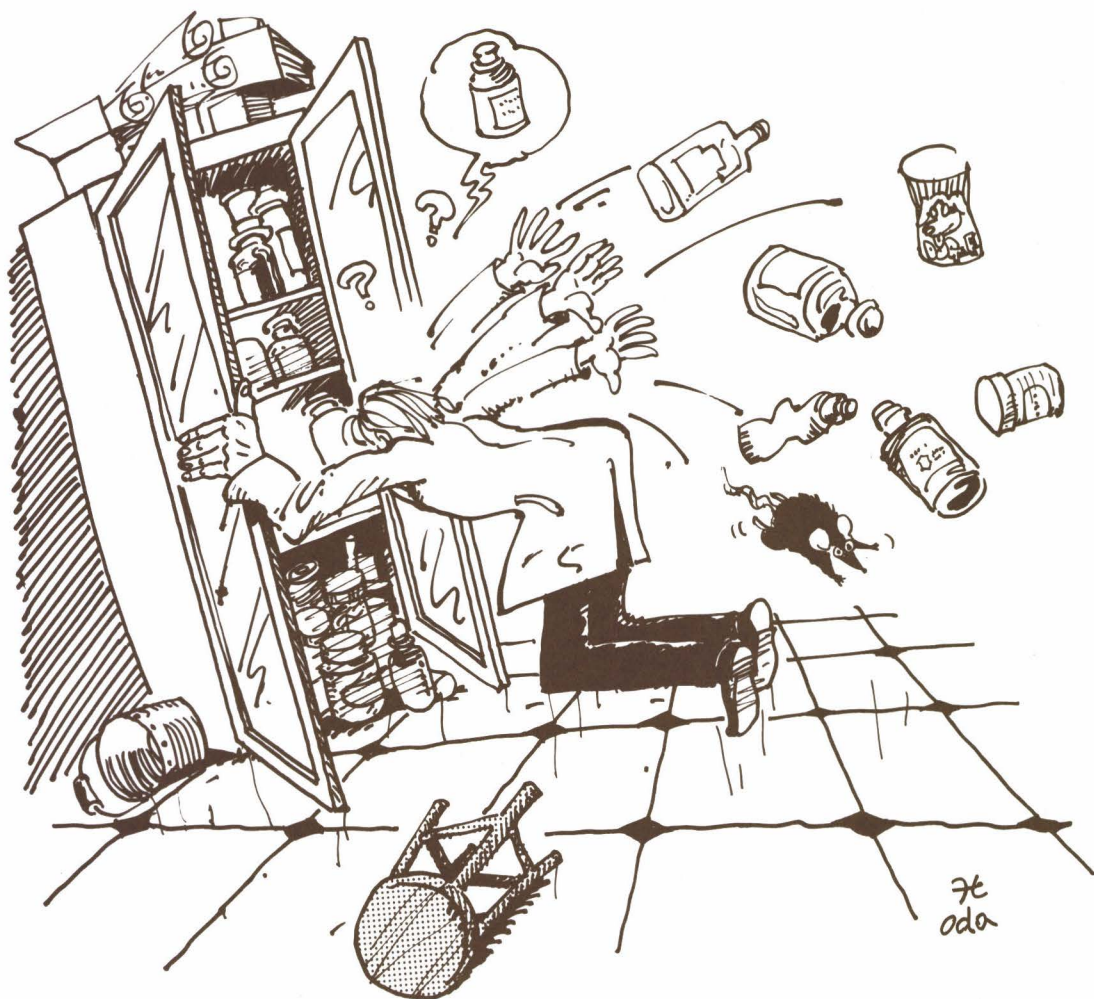
半角カタカナは，１文字２バイトなので２バイト出力して，桁位置を１つ増やす（143～144行, 151～152行）

禁則文字の場合は改行せずに，そのままその文字を出力する（156～160行）

TABコードの処理（165～173行）

```
190:     iniktype ();
191:     while (--argc && **++argv == '-') {
192:         while (*++*argv) {
193:             if (isdigit (**argv)) {
194:                 if ((Len = atoi (*argv)) == 0) {
195:                     usage ();
196:                 }
197:                 while (isdigit (*(*argv + 1))) {
198:                     ++*argv;
199:                 }
200:                 continue;
201:             }
202:             switch (**argv) {                         オプション，改行幅の処理
203:             case 'a':
204:                 Aflag = TRUE;
205:                 continue;
206:             case 'k':
207:                 Kflag = TRUE;
208:                 *argv = addkin (*argv + 1) - 1;
209:                 break;
210:             default:
211:                 usage ();
212:             }
213:         }
214:     }
215:     ++argc;
216:     --argv;
217:     if ((p = getenv ("KINSOKU")) != NULL) {   環境変数"KINSOKU"が設定されてい
218:         Kflag = TRUE;                          れば，それらの文字を禁則文字として
219:         addkin (p);                            登録する
220:     }
221:     if (Aflag) {
222:         Kflag = FALSE;
223:         uniniktp ();
224:     }
225:     if (argc == 1) {
226:         if (isatty (fileno (stdin))) {
227:             usage ();
228:         }
229:         fold (stdin);
230:         exit (0);
231:     }
232:     while (--argc) {
233:         if ((fp = fopen (*++argv, "r")) == NULL) {
234:             fputs (Prgnam, stderr);
235:             fputs (": can't open ", stderr);
236:             fputs (*argv, stderr);
237:             fputs ("¥n", stderr);
238:             exit (1);
239:         }
240:         fold (fp);
241:         fclose (fp);
242:     }
243:     exit (0);
244:
```

リスト3-13　JFOLD3.C(EUC版)

第4章
MS-DOS上での
プログラミング

Cの言語仕様は，ほかの高級言語に比べると比較的小さいため，コンパイラの移植はそれほど難しくありません．そのため，UNIXだけでなくほかのOSでも一般的に利用されるようになりました．とくにMS-DOS上では，ハードウェアの高性能化と市場の大きさに対応して，各社から本格的なCの処理系がリリースされています．

ところが，ファイルシステムに関するC言語の標準ライブラリの仕様は，UNIXのファイル構造をそのまま利用しているため，MS-DOSなどのOSで同等のライブラリを実現することは非常に難しくなっています．たとえば，UNIX上の標準ライブラリであるstat関数は，UNIXのファイルシステムを前提にファイルの管理情報を返す関数なので，MS-DOSで同等のライブラリを作成しようとしても，一部の情報が取得できないなどの問題が生じます．このような場合，プログラムの移植性を犠牲にしてもMS-DOSの機能を直接利用しなければなりません．

本章では，こういったMS-DOSに固有のライブラリ関数の使用法とそのプログラミングに焦点を当て，OSに依存するプログラムの作り方を解説します．

4.1 システムコールを使ったプログラミング

この節では，とくに MS-DOS のシステムコールを直接使ったプログラムを C 言語でどのように実現するかということを解説します．だだし，システムコールそのものについては解説しませんから，予備知識のない方は MS-DOS 関連書籍をご覧になることをお勧めします．

■ intdos関数とは

MS-DOS 上で C のプログラムを作成する際，処理系の標準ライブラリでは用意されていない機能を利用したいことがあります．こういった場合は OS の機能(ファンクションコール)を直接呼び出して利用します．ファンクションコールを直接呼び出すと，プログラムの移植性は低下しますが，MS-DOS の機能をフルに活用したプログラムが作成できます．また，ファンクションコールの直接呼び出しは，ライブラリ関数を介して OS の機能を利用するよりも，高速になる場合が多いようです(**図 4-1**)．

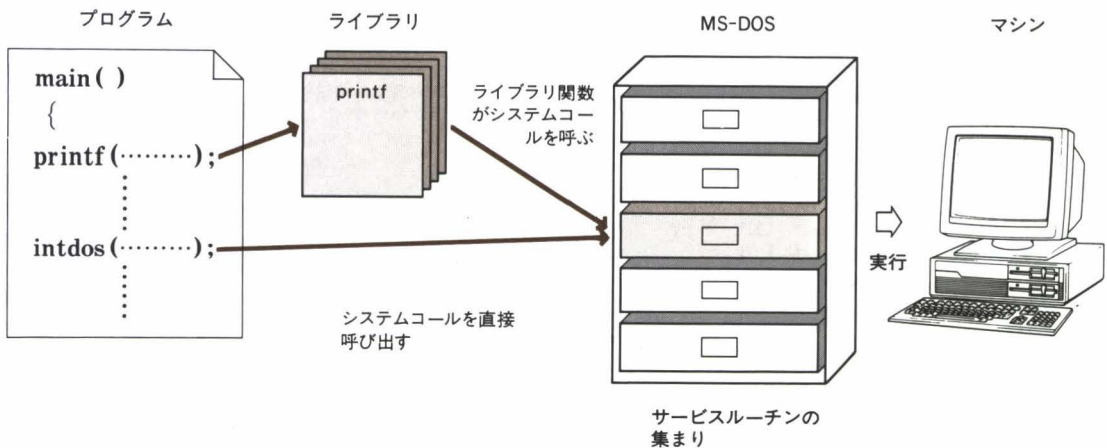



図 4-1 ファンクションコールの直接呼び出しの概念

MS-DOS 上の処理系の多くは，ファンクションコールを直接呼び出すために，独自の関数をライブラリの一部として提供しています．ここでは MS-C のライブラリのなかで，ファンクションコールを利用するための関数と，8086CPU の内部割り込みを利用するための関数について説明します．

これらの関数は，MS-C以外のほかの処理系でもほぼ同じものが提供されています．最近のコンパイラのカタログには，「MS-Cコンパチブル」をうたっているものもあります．しかし，実際には仕様が異なる場合があり，移植にあたっては注意が必要です．

本章ではMS-C以外のコンパイラについても，仕様の異なる点を表の脚注などで簡単に紹介しておきます．くわしい呼び出し方法については，各コンパイラのマニュアルを参照してください．

ファンクションコールの直接呼び出し機能を利用するプログラムは，第1章でも触れたように「DOS.H」ファイルをインクルードしておく必要があります．DOS.Hファイルでは，関連する関数の型宣言とともに，引数の受け渡しを行うための構造体や共用体の宣言が行われています．

■ データの受け渡し

C言語では，一般的に引数はスタック上に積まれて渡されますが，MS-DOSのファンクションコールでは，8086・CPUのレジスタで引数を受け渡します．しかし，MS-Cなど多くの処理系では，Cのプログラムで CPUのレジスタに明示的にアクセスすることはできません．そこで引数の受け渡しには，DOS.Hで定義されている「REGS共用体」，「SREGS構造体」という共用体／構造体を利用します．**図4-2**には，MS-Cで定義されている内容を示します．ほかの処理系でもほぼ同じような内容で定義されていますが，使用するまえにDOS.Hを参照し確認してください．

```
 1: struct WORDREGS {
 2:      unsigned int ax;
 3:      unsigned int bx;
 4:      unsigned int cx;
 5:      unsigned int dx;
 6:      unsigned int si;
 7:      unsigned int di;
 8:      unsigned int cflag;
 9:      };
10:
11: struct BYTEREGS {
12:      unsigned char al, ah;
13:      unsigned char bl, bh;
14:      unsigned char cl, ch;
15:      unsigned char dl, dh;
16:      };
17:
18: union REGS {
19:      struct WORDREGS x;
20:      struct BYTEREGS h;
21:      };
```

WORDREGS……処理系によって構造体／共用体のタグ名が異なっている場合がある

unsigned int cflag;……フラグレジスタの内容が返る構造体のメンバー．Lattice Cでは，このメンバーがなく，関数の返値としてフラグレジスタの内容が返る

```
22:
23: struct SREGS {
24:       unsigned int es;
25:       unsigned int cs;
26:       unsigned int ss;
27:       unsigned int ds;
28:       };
```

図 4-2 REGS 共用体／SREGS 構造体の中身(抜粋)

■ ファンクションコールを呼び出すライブラリ関数

ファンクションコールを直接呼び出すための関数は3種類用意されており，必要に応じて最適なものを選べるようになっています．**表 4-1** にファンクションコールを呼び出す関数を示します．

関数名	書　式	返 値	機　能
intdos()	int intdos(inregs, outregs) ; union REGS *inregs ;……ファンクションコール呼び出し時のレジスタの値 union REGS *outregs ; …ファンクションコールから戻った時のレジスタの値の格納先	AXレジスタ[†1]	MS-DOSのファンクションコールを呼び出す. セグメントは指定できない
intdosx()[†2]	int intdosx(inregs, outregs, sregs) ; union REGS *inregs ; ……ファンクションコール呼び出し時のレジスタの値 union REGS *outregs ; …ファンクションコールから戻った時のレジスタの値の格納先 struct SREGS *sregs ; …セグメントレジスタの値	AXレジスタ[†1]	MS-DOSのファンクションコールを呼び出す. セグメントの指定が可能
bdos()	int bdos(funcno, dxreg, alreg) ; int funcno ; ……………………… DOSファンクションの番号 unsigned int dxreg ;………DXレジスタの値 unsigned int alregs ; ………ALレジスタの値	AXレジスタ	MS-DOSのファンクションコールを呼び出す. AX/DXレジスタを指定し，結果はAXレジスタのみが返る

†1 : Lattice Cではフラグレジスタの値が返る
†2 : Lattice C, LSI C-86の場合，セグメントレジスタの値を取得することができない(設定は可能)
　　セグメントレジスタの値を取得するためには，intdoss(Lattice Cの場合)，intdosy(LSI C-86の場合)の各関数を用いる

表 4-1 ファンクションコールを呼び出すためのライブラリ関数

intdos 関数は，引数 inregs の指す共用体の値をレジスタにセットして，MS-DOS のファンクションコールを呼び出します．ファンクションコールから戻ってきたときのレジスタの値は，outregs で指定された共用体に格納されます．この関数ではセグメントレジスタの値を指定する必要はなく，intdos 関数が呼び出された時点でのセグメントレジスタの値がそのまま使われます．8086 用のメモリモデルのうちスモールモデルを利用している場合は，この intdos 関数を利用すればセグメントレジスタについて気にする必要はありません．

intdosx 関数は，intdos 関数の機能を包含した関数であり，さらにセグメントレジスタの値も受け渡すことができます．セグメントレジスタの値は，sregs で指定される構造体にセットされ，ファンク

ションコールから戻ってきたときの値が，再び構造体に格納されます．スモールモデルを利用している場合でも，ESレジスタの値を参照するファンクションコールを呼び出す場合には，intdos関数ではなく，intdosx関数を利用しなければなりません*1．

bdos関数は，上述の2つの関数と異なり，呼び出し時にはAXレジスタとDXレジスタの値しか指定できず，戻ってきた時点でのAXレジスタの値しか得ることができません．MS-DOSのファンクションコールの大部分は，エラーが起きた場合にキャリーフラグ(REGS共用体のメンバーcflag)をセットして戻ってきますから，エラーの起きる可能性のあるファンクションコールは，このbdos関数で呼び出すと危険です．ただし，AXレジスタとDXレジスタの値だけ渡せばよく，エラーが絶対に起きない場合は，bdos関数を利用して簡単にファンクションコールを呼び出すことができます．

■ 内部割り込みを起動する関数

第1章で説明したように，8086・CPUではINT命令と呼ばれる命令が用意されており，MS-DOSの各種の機能を呼び出す場合などに利用されます．たとえば，先に説明したファンクションコールも「INT 21h」という命令によって起動されます．

INT命令は，「0h」から「FFh」まで，256種類の割り込みベクタを指定することができます．MS-Cでは，これらの内部割り込みを起動するためのライブラリ関数が用意されています．すでにサンプルプログラムは第1章でも紹介しましたが，ここで**表4-2**にあらためてまとめておきます．

int86関数とint86x関数は，割り込みベクタを指定できる点を除くとintdos関数，intdosx関数と同じ仕様になっています．もちろん，割り込みベクタとして「0x21」を指定すれば，int86/int86x関数でintdos/intdosx関数を代用することもできます(多少遅くなりますが)．

関数名	書　式	返　値	機　能
int86()	int int86(intno, inregs, outregs) ; int intno ; ……割り込みベクタ union REGS *inregs ; ……ファンクションコール呼び出し時のレジスタの値 union REGS *outregs ; ……ファンクションコールから戻った時のレジスタの値の格納先	AXレジスタ†1	引数intnoで指定された内部割り込みを実行する．セグメントは指定できない
int86x()†2	int int86x(intno, inregs, outregs, sregs) ; int intno ; ……割り込みベクタ union REGS *inregs ; ……ファンクションコール呼び出し時のレジスタの値 union REGS *outregs ; ……ファンクションコールから戻った時のレジスタの値の格納先 struct SREG *sregs ; ……セグメントレジスタの値	AXレジスタ†1	引数intnoで指定された内部割り込みを実行する．セグメントの指定が可能

†1 : Lattice Cではフラグレジスタの値が返る
†2 : Lattice C, LSI C-86の場合，セグメントレジスタの値を取得することができない(設定は可能)
　セグメントレジスタの値を取得するためには，int86s(Lattice Cの場合)，int86y(LSI C-86の場合)の各関数を用いる

表4-2　内部割り込みを実行するためのライブラリ関数

*1　MS-DOSのファンクションコールについては，「MS-DOS3.1ハンドブック」(アスキー出版局編著　アスキー出版局発行)などの書籍を参照のこと．

■ 現在のセグメントレジスタの値を得る関数

intdosx 関数や int86x 関数を利用して，内部割り込みを実行する場合には，常にセグメントレジスタの値を設定する必要があります．しかし，特定のセグメントレジスタ以外は，現在のレジスタの値をそのまま渡したい場合，なんらかの方法でその値を SREGS 構造体に読み込み，その後に変更したいレジスタの値のみを書き換える必要があります．

segread 関数は，現在のセグメントレジスタの値を，引数で指定された SREGS 構造体に読み込む関数です．**表 4-3** に segread 関数の仕様を示します．

関数名	書　式	返 値	機　能
segread()	void segread(sregs) ; struct SREGS *sregs ;	なし	現在のセグメントレジスタの値を，指定されたSREGS 構造体に読み込む

表 4-3　segread 関数の仕様

■ intdos関数の必要性

ファンクションコールを直接呼び出す(intdos 関数を使う)とプログラムは MS-DOS でしか動作しなくなりますから，使わずにすめば，それにこしたことはありません．しかし，次のような場合には，intdos 関数を利用せざるをえません．

・ライブラリ関数では提供されていない機能を利用したい場合
・速度を少しでも向上させたい場合

実際には，第 1 の理由がファンクションコールをそのまま利用する最大の理由です．MS-DOS には用意されているのにライブラリ関数では用意されていない機能を利用したい場合には，必然的に int-dos 関数を利用することになります．たとえば，ディレクトリの内容を読み出す関数は MS-C には用意されていないので，ファンクションコールを直接呼び出して実現する必要があります．

最近では，これらのシステムコールを直接呼び出す関数を別のライブラリにまとめていることも多いようです(Turbo C など)．これらの新しい関数は使いやすいのですが，ほかの処理系とのコンパチビリティが失われることが多く，あまりお勧めできません．

4.2　ディレクトリを検索するシステムコール

ここでは，MS-DOSのファンクションコールを直接利用しないと実現できないプログラムを作ることで理解を深めましょう．サンプルとしてMS-Cのintdosライブラリを利用してプログラムを作成することにします．

MS-DOSの機能の大部分は，Cの処理系に付属する標準関数を介して利用することができます．しかし次に示す機能などは，MS-DOSとUNIXというオペレーティングシステムの構造上の違いが大きく，MS-DOSではファンクションコールを直接利用しなくてはなりません．

・ディレクトリの検索
・ファイル属性の取得／設定
・カレントドライブの取得／変更

UNIXはMS-DOSとは違ってカレントドライブなどというものがなく，すべてのデバイスやファイルが統一されたファイルシステムの上に成り立っています．これは既存のファイルを消すというライブラリ関数である「unlink()」などの名称が，どういう経過でできたかという考え方の違いをみると明らかです．少し立ち入った話になりますが，UNIXのファイルシステムではunlinkという名前のとおり，iノードのリンクを切りフリーエリアのリンクに加えることによってファイルを消去するわけですが，MS-DOSではファイルのディレクトリエントリに消去マークをつけることになります．

■ ディレクトリの検索　—LDコマンド—

ディレクトリを検索するプログラムの一例として，ディレクトリの内容を表示するLD(List Directory)というコマンドを作ってみることにしましょう．LDコマンドはDIRコマンドと同じ働きをするものですが，各ファイルについての情報をよりくわしく表示させることができます．

ディレクトリの内容を検索するためには，ファンクション4Eh(FINDFIRST)とファンクション4Fh(FINDNEXT)を利用します．また，関連する機能として，ファンクション1Ah(SETDTA)があります．次ページの**表4-4**に，これらのファンクションコールの仕様を示します．

FINDFIRSTとFINDNEXTは，組みにして利用するファンクションコールです．最初の検索にFINDFIRSTを，2度目以降の検索にFINDNEXTを使います．FINDFIRSTには，検索条件としてサーチするファイル名とファイル属性を指定します．該当するエントリがあれば，そのエントリに格納されている情報が，DTA(Disk Transfer Address)領域と呼ばれる領域に格納されます*2．DTA領

---

*2　DTAについては，MS-DOS関連書籍を参照のこと．

ファンクション	ファンクション番号	コール手順	返　値	機　　能
SETDTA	1Ah	ds : dx＝DTAのアドレス ah＝1Ah INT 21h	なし	ds : dx で指定されたアドレスを，DTAのアドレスとして登録する
FINDFIRST	4Eh	ds : dx＝パスのアドレス cx＝検索属性 ah＝4Eh INT 21h	キャリーフラグにエラーの有無が返る	ds : dx と cx で指定されたファイルを検索し，発見できた場合にはそのファイルに関する情報をDTAに格納する
FINDNEXT	4Fh	ah＝4Fh INT 21h	キャリーフラグにエラーの有無が返る	FINDFIRST が初期化したDTA領域の内容にしたがって，次のファイルを検索する．発見できた場合には，そのファイルに関する情報をDTA領域に格納する

表 4-4　SETDTA，FINDFIRST，FINDNEXT の仕様

域のアドレスは，FINDFIRST の引数として指定するのではなく，SETDTA によってあらかじめ設定しておく必要があります．

FINDFIRST で，検索ファイル名としてワイルドカード（〝?〞，〝*〞など）を含むパスを指定した場合，複数のエントリが検索条件に合致する可能性があります．このような場合には，FINDNEXT を呼び出すと，2つ目以降のエントリに関する情報が得られます．FINDFIRST で指定した検索条件は，DTA の領域に格納されていますから，FINDNEXT に引数はありません（DTA のアドレスを FINDFIRST を呼び出したときと同じようにセットする必要はある）．

これらのファンクションコールをCから利用できるように，setdta 関数，findfirst 関数，findnext 関数の3つを作成します．ここでは，findfirst 関数と findnext 関数の引数として DTA 領域のアドレスを指定できるようにして，プログラムの上位ルーチンからは，setdta 関数を直接呼び出す必要がないようにしています．**表 4-5** にこれらの関数の仕様を，**リスト 4-1** と**リスト 4-2** にプログラムを示し

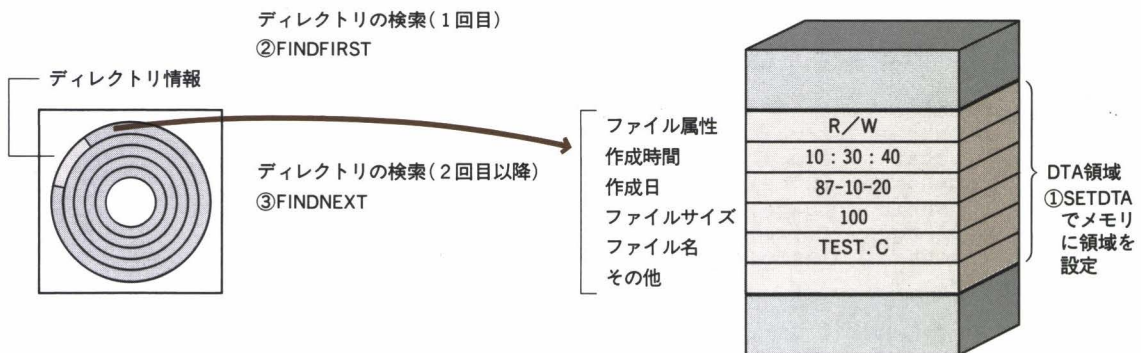



図 4-3　FINDFIRST と FINDNEXT によるディレクトリの検索

ます．「DIR.H」は DTA 領域に格納される情報を，C のプログラムから参照するための構造体の定義です．

関数名	書　式	返 値	機　能
setdta()	void setdta(dta); char *dta; ……DTAのアドレス	なし	引数で指定されたアドレスをDTAとして登録する
findfirst()	int findfirst(path, attr, dta); char *path; ……検索するファイル名 unsigned short attr; ……検索属性 DIR *dta; ……ファイル情報を格納するバッファ	−1＝エラー 0＝成功	pathで指定されたパスを検索し，ファイル名と属性の条件を満たすファイルの情報をDTAで指定された構造体に格納する
findnext()	int findnext(dta); DIR *dta; ……ファイル情報を格納するバッファ	−1＝エラー 0＝成功	findfirst関数が初期化したDTAの内容にしたがって，条件を満たす次のファイルを検索し，そのファイルの情報をDTAに格納する

表 4-5　setdta 関数，findfirst 関数，findnext 関数の仕様

```
 1: /*
 2:  *  dir.h: struct definition for findfirst()/ findnext()
 3:  */
 4:
 5: typedef unsigned char   uchar;
 6: typedef unsigned short  ushort;
 7: typedef unsigned long   ulong;
 8: typedef unsigned int    uint;
 9:
10: /* ------- bit definitions of attribute byte ------------ */
11:
12: #define DIR_RO  0x0001          /* read only       */
13: #define DIR_HID 0x0002          /* hidden file     */
14: #define DIR_SYS 0x0004          /* system file     */
15: #define DIR_VOL 0x0008          /* volume name     */
16: #define DIR_DIR 0x0010          /* directory entry */
17: #define DIR_ARC 0x0020          /* archive bit     */
18:
19: /* ------- structure definitions of DIR structure ------ */
20:
21: typedef struct {
22:     char    _reserved[21];      /* reserved area    */
23:     uchar   _attrib;            /* attribute byte   */
24:     ushort  _ftime;             /* time of file     */
25:     ushort  _fdate;             /* date of file     */
26:     ulong   _fsize;             /* file size        */
27:     uchar   d_name[13];         /* packed file name */
28: } DIR;
29:
30: /* ---- end of dir.h ---- */
```

リスト 4-1　DIR.H

```
 1: /*
 2:  *  dosfind.c:  MS-DOS Interface for search directory entry
 3:  */
 4:
 5: #include <dos.h>
 6: #include <stdio.h>
 7: #include "dir.h"
 8:
 9: union REGS regs;
10:
11: void    setdta(DIR *p)
12: {
13:     regs.x.dx = (ushort)p;
14:     regs.h.ah = 0x1a;
15:     intdos(&regs, &regs);
16: }
17:
18: int findfirst(char *path, ushort attrib, DIR *dta)
19: {
20:     setdta(dta);
21:     regs.x.cx = attrib;
22:     regs.x.dx = (ushort)path;
23:     regs.h.ah = 0x4e;
24:     intdos(&regs, &regs);
25:     return regs.x.cflag ? -1 : 0;
26: }
27:
28: int findnext(DIR *dta)
29: {
30:     setdta(dta);
31:     regs.h.ah = 0x4f;
32:     intdos(&regs, &regs);
33:     return regs.x.cflag ? -1 : 0;
34: }
35:
36: /* ---- end of dosfind.c ---- */
```


リスト 4-2　DOSFIND.C

DTA 領域には，ディレクトリエントリの情報がひと通り格納されていますから，ファイル名やその大きさ，ファイル属性，タイムスタンプなどの情報を利用することができます．この領域には，それ以外にもさまざまなファイルの管理情報が格納されていますが，それらの領域についてはいっさいアクセスしてはいけません．以下に，各メンバーに格納されている情報について説明します．

d_name

検索されたエントリのファイル名が格納されています．このメンバーは，C などから利用しやすいように，文字列の最後が「¥0」で終わっており，そのまま C の文字列としてアクセスすることができます．

_fsize

ファイルの大きさが格納されています．ただし，検索されたエントリがディレクトリだった場合には，このメンバーには常に 0 が格納されています．

_ftime ／_fdate

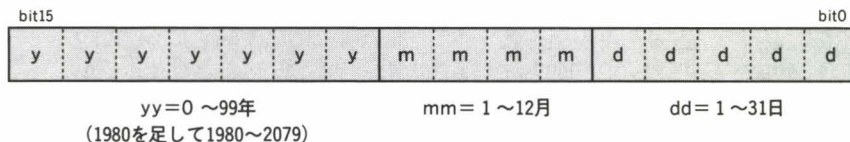
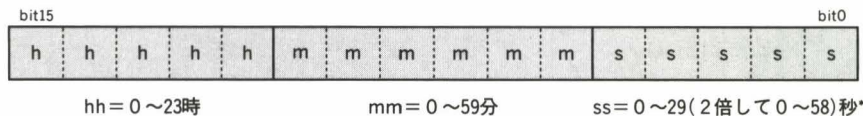
ファイルのタイムスタンプが格納されています．UNIX や C 言語の標準ライブラリでは，時間情報は原則として 1970 年の 1 月 1 日 0 時 0 分 0 秒からの通算秒として表しますが，MS-DOS ではタイムスタンプは 1980 年 1 月 1 日 0 時 0 分 0 秒を基点に，ビットフィールドの形で格納されています(**図 4-4**)．

＊MS-DOSでは，2 秒単位でしか時間を記録できないので，秒の値は1/2にして格納される

図 4-4　MS-DOS のタイムスタンプ

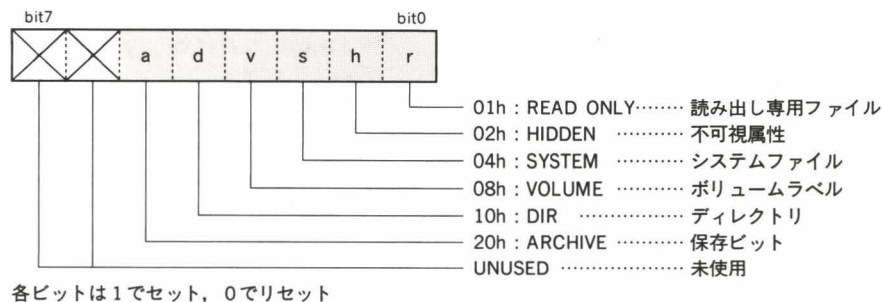



図 4-5　ファイル属性のフォーマット

_attrib

ファイル属性が格納されています．データのフォーマットは図4-5のとおりです．

このファイル属性は，FINDFIRSTでも指定する必要がありますから，各ビットの意味やその利用方法について，もう少しくわしく説明しておきましょう．

ビット0(R／Oビット)… 読み出し専用ファイル

このビットがセットされているファイルは読み出し専用ファイルであり，書き込みの対象としてオープンすることはできません．また，そのファイルを削除することもできません．たとえば，読み出し専用ファイルをDELコマンドで削除しようとすると，「ファイルが見つかりません」,「アクセスは拒否されました」などのエラーになってしまいます．

ビット1(hiddenビット)… 不可視ファイル
ビット2(systemビット)… システムファイル

これらのビットがセットされているファイルは，DIRコマンドでは表示されません．systemビットはMS-DOSにとってとくに重要なファイルにのみつけられ，一般ユーザーに見せないようにするために利用されます．

ビット3(volumeビット)… ボリュームラベル

MS-DOSでは，ディスクの各メディアにボリュームラベルをつけることができます(DIRコマンドやVOLコマンドで表示される)が，そのボリュームラベルは，ルートディレクトリの特殊なエントリとして実現されています．そのエントリにのみこのvolumeビットがセットされます．つまり，このビットの立っているエントリは，ファイルでもディレクトリでもありません．

ビット4(dirビット)… ディレクトリエントリ

そのエントリが，ディレクトリ名を表すことを意味します．

ビット5(archビット)… アーカイブ・ビット

ファイルが作成されたり，その後そのファイルに書き込みが行われた場合に，MS-DOSによって自動的にセットされます．このビットはプログラムが明示的にリセットしないかぎり，セットされたままになっており，バックアップ用のコマンド(Ver3.1以上に付属するBACKUPコマンドなど)に利用されます．

FINDFIRSTファンクションコールで検索条件として指定する場合には，これらのビットのうち，R/Oビットとarchビットは無視されます．つまり，R/Oビットとarchビットは検索条件として指定

しても意味がありません．また，ビット6とビット7について，マニュアルではとくに規定はありません．

findfirst関数とfindnext関数を利用したLDコマンドを**リスト4-4**に示します．このLDコマンドは，指定されたファイルを検索して，見つかったファイルの管理情報を表示します．また，指定されたディレクトリエントリが見つからない場合には，エラーを表示します．findfirst関数とfindnext関数は，ワイルドカードの展開機能を持っていますから，ワイルドカードも指定できます．

```
 1: dosfind.obj: dir.h dosfind.c
 2:     cl -AS -c dosfind.c
 3:
 4: ld.obj: dir.h ld.c
 5:     cl -AS -c ld.c
 6:
 7: ld.exe: ld.obj dosfind.obj
 8:     cl -Feld.exe ld.obj dosfind.obj
```

リスト4-3　MAKEFILE

```
 1: /*
 2:  *   ld.c: test program for findfirst()/findnext()
 3:  */
 4:
 5: #include <dos.h>
 6: #include <stdio.h>
 7: #include <stdlib.h>
 8: #include <string.h>
 9: #include "dir.h"
10:
11: typedef int BOOL;
12: #define TRUE     1
13: #define FALSE    0
14:
15: BOOL     Vflag = FALSE;
16:
17: #define YEAR(d)        ((((d) >> 9) & 0x7f) + 80)
18: #define MONTH(d)       (((d) >> 5) & 0x0f)
19: #define DATE(d)        ((d) & 0x1f)
20: #define HOUR(d)        (((d) >> 11) & 0x1f)
21: #define MINUTE(d)      (((d) >> 5) & 0x3f)
22: #define SEC(d)         (((d) & 0x1f) * 2)
23:
24: showdata(DIR *dta)
25: {
26:     uchar   mask;
```

（17〜22行：タイムスタンプを分解して，各要素を取り出すマクロ）
（24行：各ディレクトリエントリの内容を表示する関数）

```
27:
28:     for (mask = 0x80; mask != 0; mask >>= 1)
29:         putchar(dta->_attrib & mask ? '1' : '0');
30:     printf(" %8lu %02d/%02d/%02d %02d:%02d:%02d %s¥n",
31:         dta->_fsize,
32:         MONTH(dta->_fdate), DATE(dta->_fdate), YEAR(dta->_fdate),
33:         HOUR(dta->_ftime), MINUTE(dta->_ftime), SEC(dta->_ftime),
34:         dta->d_name);
35: }
36:
37: ls(char *path)……………………指定されたパスを検索する関数
38: {
39:     ushort  attrib;
40:     DIR dta;
41:
42:     if (Vflag)
43:         attrib = 0x003f;
44:     else                                  オプションによって検索属性を変更する
45:         attrib = 0x003f & ~DIR_VOL;
46:     if (findfirst(path, attrib, &dta) == -1) {  1つもエントリが見つからない場合は，エラーを表示
47:         fprintf(stderr,"ld: %s: No such file or directory.¥n",path);
48:         return -1;
49:     }
50:     do {  ……………………各エントリの処理を行うループ
51:         showdata(&dta);
52:     } while (findnext(&dta) != -1); ……………………findnext関数がエラーになるまで繰り返す
53:     return 0;
54: }
55:
56: usage()
57: {
58:     fprintf(stderr, "Usage: ld [-V] [<path>...]¥n");
59:     exit(1);
60: }
61:
62: main(int argc, char **argv)
63: {
64:     char    *p;
65:     int error;
66:
67:     error = 0;
68:     while (--argc > 0 && **++argv == '-') {
69:         if (strlen(*argv) == 1) {
70:             --argc;
71:             ++argv;
72:             break;
73:         }
74:         for (p = *argv + 1; *p != '¥0'; p++) ……………………オプションの解析部
75:             switch (*p) {
76:             case 'V': ……………………"V"を指定するとボリュームビットもセットされる
77:                 Vflag = TRUE;
78:                 break;
79:             default:
80:                 usage();
81:                 /*NOTREACHED*/
82:             }
83:     }
```

```
84:     if (argc == 0) { ……………検索パスを指定しない場合，"*.*"を検索する
85:         if (ls("*.*") != 0)
86:             error++;
87:     } else
88:         while (argc-- > 0)
89:             if (ls(*argv++) != 0)
90:                 error++;
91:     exit((error == 0) ? 0 : 2);
92: }
93:
94: /* ---- end of ld.c ---- */
```

[実行結果]

```
B>ld .
00010000          0 10/01/90 08:46:36 LD
└ファイル属性┘└サイズ┘└ 作成日 ┘└作成時間┘└ファイル名┘
B>ld ¥
ld: ¥: No such file or directory.   Ver3.1以降ではルートディレクトリの検索にはかならず失敗する．
                                    Ver2.11では，正しい内容が返らない．

B>ld ld.exe
00100000       8145 10/01/90 08:53:22 LD.EXE

B>ld con
01000000       4521 10/01/90 08:56:28 CON   Ver3.1以降ではキャラクタデバイス名の検索が可能
                                            だが，Ver2.11では，正しい内容が返らない．

B>
```

リスト4-4　LD.C

■ MS-DOSのバージョンによる動作の違い

MS-DOSにはいろいろなバージョンがあります．日本では主に現在Ver3.3以降のものが売られていますが，実際の環境では，Ver2.11やVer3.1などが使われている場合もあります．MS-DOSのマニュアルによるとFINDFIRSTとFINDNEXTは，2つのバージョンのどちらでも仕様が同じなので，これらのファンクションコールを利用したプログラムは，どのバージョンのMS-DOSでも，まったく同じように動作するはずです．

ところが，このLDコマンドをおのおののバージョンのMS-DOSで動作させてみると，いくつかの点で，その動作に違いがあることがわかります．

Ver2.11では，検索ファイル名として「¥」(ルートディレクトリ)を指定すると，FINDFIRSTはエラーを返しませんが，DTA領域の内容はいいかげんなもので，その情報を信用することはできません．Ver3.1以降では，ルートディレクトリの検索にはかならず失敗します．

またVer3.1では，キャラクタデバイス名(CON，PRNなど)をファイル名として指定しても正常に動作しますが，Ver2.11では正しい情報を返しません．Ver3.1以降では検索条件として指定するファ

イル属性のボリュームラベルビットが意味を持っており，ボリュームラベルビットをセットすると(-V オプションで指定する)，ボリュームラベルは検索できますが，キャラクタデバイスは検索できなくなります．逆にリセットしておくと，キャラクタデバイスは検索できますが，ボリュームラベルは検索できません．

さらに，カレントディレクトリがサブディレクトリのとき，検索ファイル名として「.」を指定すると，Ver2.11 ではカレントディレクトリの「.」のエントリの内容が返されますが，Ver3.1 以降では親ディレクトリからそのディレクトリを指しているエントリの内容が返されます(LD コマンドを実行するとディレクトリ名が表示される)．

このように MS-DOS では，マニュアルにはいっさい明記されないままに，バージョンごとにファンクションコールの仕様が変更されている場合があり，注意が必要です．

■ 指定されたパスの属性を得る

MS-DOS でディレクトリを検索するプログラムを作成すると，あるパスがファイルなのかディレクトリなのか，ディレクトリであればルートディレクトリなのかサブディレクトリなのかといった情報が必要な場合がでてきます．FINDFIRST を利用すれば，そのパスがディレクトリなのかどうかは判断できますが，ルートディレクトリの扱いが貧弱なため，そのままではルートディレクトリかどうかのチェックができません．また前節で述べたように，Ver2.11 では，ルートディレクトリの可能性があるパスは，そのまま FINDFIRST に渡すことができません．

このような場合，ルートディレクトリかどうかは，指定されたパスにカレントディレクトリを変更してみて，さらにその親ディレクトリに移動できるかどうかによって判断します．以下に紹介するルーチン，filetype 関数は，指定されたパスを検索してそのパスの種類を返す関数です．**表 4-6** に filetype 関数の仕様を，**リスト 4-5～リスト 4-8** にプログラムを示します．

関数名	書　式	返　　値	機　能
filetype()	int filetype(path); char *path;	PATH_ERROR …………ドライブ名が不正 PATH_AMBIGUOUS ……引数にワイルドカードが含まれている PATH_NONE ……………存在しない PATH_ROOTDIR ………ルートディレクトリである PATH_SUBDIR …………サブディレクトリである PATH_REGULAR ………一般ファイルである	引数pathで指定されたパスの有無と，そのタイプを返す

表 4-6　filetype 関数の仕様

```
 1: /*
 2:  *  filetype.h:
 3:  */
 4:
 5: /* -------- return value definition of filetype() -------- */
 6:
 7: #define PATH_ERROR      (-2)     /* non exist drive */
 8: #define PATH_AMBIGUOUS  (-1)     /* contain wild-card */
 9: #define PATH_NONE       (0)      /* unexist path. */
10: #define PATH_ROOTDIR    (1)      /* root directory. */
11: #define PATH_SUBDIR     (2)      /* sub directory. */
12: #define PATH_REGULAR    (3)      /* normal file. */
13:
14: /* ---- end of filetype.h ---- */
```

リスト4-5　FILETYPE.H

```
 1: /*
 2:  *  filetype.c: library functions for directory access.
 3:  */
 4:
 5: #include <signal.h>
 6: #include <dos.h>
 7: #include <ctype.h>
 8: #include <stdio.h>
 9: #include <string.h>
10: #include "dir.h"
11: #include "filetype.h"
12:
13: typedef int BOOL;
14: #define TRUE     (1)
15: #define FALSE    (0)
16: #define SUCCESS  (TRUE)
17: #define FAIL     (FALSE)
18:
19: /*
20:  *  getcdir(): get current directory of specified drive.
21:  *  ret = getcdir(drive, buf);
22:  *  uchar  drive;  1 for A, 2 for B ...
23:  *  char   *buf;   buffer to save result
24:  *  BOOL   ret;    FAIL for error, SUCCESS for success.
25:  */
26:
27: static  BOOL    getcdir(uchar drive, uchar *buf)
28: {
29:     union REGS regs;
30:
31:     regs.h.dl = (uchar)drive;
32:     regs.x.si = (uint)buf;
33:     regs.h.ah = (uchar)0x47;      /* get current dir */
34:     intdos(&regs, &regs);
35:     if (regs.x.cflag)          /* Illegal drive number */
```

……指定されたドライブのカレントディレクトリを取得する関数(先頭のドライブ名と"¥"は付かない)

```
36:         return FAIL;
37:     return SUCCESS;
38: }
39:
40: /*
41:  *  _chdir(): change current directory (keep current drive).
42:  *  ret = _chdir(path);
43:  *  int     ret;    -1 if error; 0 if success;
44:  *  char    *path;  path to set.
45:  */
46:
47: static  int _chdir(char *path)……………カレントディレクトリを変更する関数
48: {                                  (カレントドライブは変更されない)
49:     union REGS regs;
50:
51:     regs.x.dx = (uint)path;………………指定されたパスの有無／種別を返す関数
52:     regs.h.ah = 0x3b;
53:     intdos(&regs, &regs);
54:     return regs.x.cflag ? -1 : 0;
55: }
56:
57: /*
58:  *  filetype(): check filetype.
59:  *  ret = filetype(path);
60:  *  int     ret;    (defined in filetype.h)
61:  *  char    *path;  path to check.
62:  */
63:
64: int filetype(char *path)
65: {
66:     DIR dtabuf;
67:     int (*func)();
68:     uchar   drive;      /* 1 for A:, 2 for B:, .... */
69:     char    cwd[68];
70:     char    parent[5];
71:     char    *p;
72:     int type;
73:     char    *jstrmatch();
74:
75:     if (jstrmatch(path, "*?[") != NULL)……………ワイルドカードを含むパスは
76:         return PATH_AMBIGUOUS;                調べられない
77:     if (isalpha(path[0]) && path[1] == ':') {  ┐
78:         parent[0] = *path;                     │
79:         parent[1] = ':';                       │
80:         strcpy(parent + 2, "..");              │
81:         drive = toupper(*path) - 'A' + 1;      │
82:         cwd[0] = *path;                        │
83:         cwd[1] = ':';                          │
84:         cwd[2] = '¥¥';                         ├ カレントディレクトリの保存
85:         p = cwd + 3;                           │
86:     } else {            /* current drive */    │
87:         strcpy(parent, "..");                  │
88:         drive = 0;                             │
89:         cwd[0] = '¥¥';                         │
90:         p = cwd + 1;                           │
91:     }                                          ┘
```

```
 92:     if (!getcdir(drive, p))
 93:         return PATH_ERROR;
 94:     func = signal(SIGINT, SIG_IGN); ···············CTRL+Cの割り込み禁止
 95:     if (_chdir(path) != 0) {/* cannot chdir; path isnot a dir */
 96:         type = (findfirst(path, 0x003f & ~DIR_VOL, &dtabuf) == 0)
 97:             ? PATH_REGULAR : PATH_NONE;···············指定されたパスにディレクトリを変更
 98:     } else {                                        できなければ findfirstの結果によって種別を決定
 99:         type = (_chdir(parent) == 0) ? PATH_SUBDIR : PATH_ROOTDIR;···
100:         _chdir(cwd);          /* restore default dir */
101:     }
102:     signal(SIGINT, func); ···············割り込み状態をもとに戻す
103:     return type;
104: }                                  ディレクトリの場合は，さらに上のディレクトリ
105:                                    にいけるかどうかでルートかどうかを判断し，······
106: /* ---- end of filetype.c ---- */  もとのディレクトリへ戻る
```

リスト 4-6　FILETYPE.C

```
  1: /*
  2:  *  chktype.c: test program for filetype()
  3:  */
  4:
  5: #include <stdio.h>
  6: #include <stdlib.h>
  7: #include "filetype.h"
  8:
  9: chktype(char *path)
 10: {
 11:     char    *p;
 12:
 13:     switch (filetype(path)) { ···················filetype関数の返値にしたがって表示を行う
 14:     case PATH_ERROR:
 15:         p = "error";
 16:         break;
 17:     case PATH_AMBIGUOUS:
 18:         p = "ambiguous";
 19:         break;
 20:     case PATH_NONE:
 21:         p = "not exist";
 22:         break;
 23:     case PATH_ROOTDIR:
 24:         p = "root-dir";
 25:         break;
 26:     case PATH_SUBDIR:
 27:         p = "sub dir";
 28:         break;
 29:     case PATH_REGULAR:
 30:         p = "regular file";
 31:         break;
 32:     default:
 33:         abort();
 34:     }
```

```
35:       printf("%s: %s¥n", path, p);
36: }
37:
38: main(int argc, char **argv)
39: {
40:       if (argc == 1)……………………デフォルトで“.”(カレントディレクトリ)の種類を表示
41:           chktype(".");
42:       else
43:           while (--argc)
44:               chktype(*++argv);
45:       exit(0);
46: }
47:
48: /* ---- end of chktype.c ---- */
```

[実行結果]

```
B>chktype . ¥ chktype.exe *.c ↵
.: sub dir
¥: root-dir
chktype.exe: regular file
*.c: ambiguous

B>
```

リスト4-7　CHKTYPE.C

```
 1: dosfind.obj: dir.h dosfind.c
 2:       cl -AS -c dosfind.c
 3:
 4: filetype.obj: dir.h filetype.h filetype.c
 5:       cl -AS -c filetype.c
 6:
 7: chktype.obj: filetype.h chktype.c
 8:       cl -AS -c chktype.c
 9:
10: chktype.exe: chktype.obj filetype.obj dosfind.obj
11:       cl -Fechktype.exe chktype.obj filetype.obj dosfind.obj
```

リスト4-8　MAKEFILE

ここで注目してもらいたい処理は2点です．1つは，FINDFIRSTにルートディレクトリを示すパスを渡さないようにするにはどうすればよいかという点です．もう1つは，プログラムでカレントディレクトリやカレントドライブを変更する場合に，どのような処理が必要かという点です．

このルーチンでは，Ver2.11のバグ(FINDFIRSTでルートディレクトリを指定したとき，正しい情報を返さない)を避けるために，とりあえずパスをディレクトリと見なして，カレントディレクトリの変更を試みます．変更に成功した場合は，そのディレクトリがルートディレクトリかどうかを判断します．ルートディレクトリとサブディレクトリの違いの1つに，親ディレクトリ「..」があるかどうかということがありますから，親ディレクトリに移動できればそれはサブディレクトリであり，失敗すればルートディレクトリであることになります．カレントディレクトリの変更に失敗した場合，指定されたパスは存在しないか，通常のファイルであるということですから，FINDFIRSTを利用して，そのパスが存在するかどうかを調べます．この方法はあまり美しい方法ではありませんが，MS-DOSではこのようにするしかありません．

また，カレントディレクトリを変更しているあいだにCTRL+Cなどが入力されると，カレントディレクトリが変更されたままでコマンドの実行を終了してしまいますから，カレントディレクトリを変更しているあいだは，signal関数などを利用して割り込みを無視したり，カレントディレクトリなどをもとの状態に戻してから実行を終了するようなルーチンを呼び出すようにしておく必要があります．このプログラムの場合は，カレントディレクトリを変更している時間が比較的短いので，変更しているあいだは割り込みを無視するようにしています．

4.3 子プロセスの実行

MS-DOS では，プログラムから別のプログラムを起動することができます．この機能を「子プロセスの起動」といいます．これは，UNIX などのマルチタスクの場合と異なり，起動したプログラムが終了するまで親プログラムは止まっているのですが，この機能があることによって，実現できることは実に多彩です．たとえばエディタのなかから，別のコマンドを実行することもできます．また，第7章で紹介する MAKE コマンドなども，この子プロセスの起動を利用して実現されたプログラムです．

MS-C では，子プロセスを起動するためのライブラリ関数が用意されていますが，この節ではその使い方や注意点について解説します*3．

■ 子プロセスを起動する関数群

MS-C には，子プロセスを起動するためのライブラリ関数が 13 個用意されていますが，これらは 3 種類に分類することができます(図 4-6)．

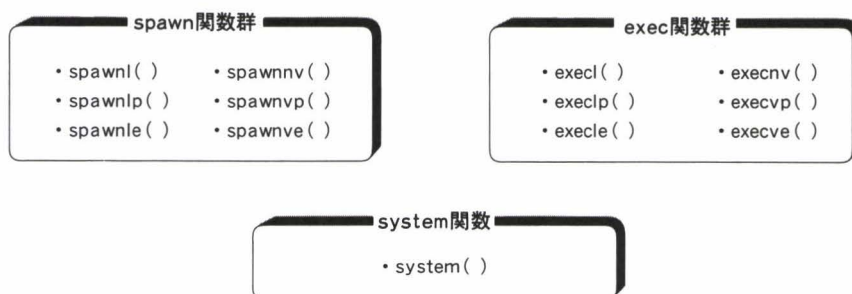
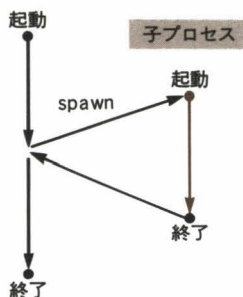



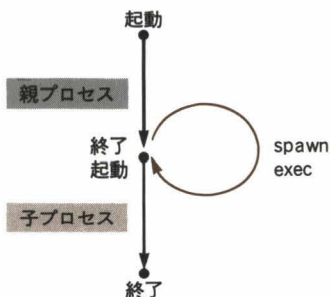
図 4-6 子プロセスを起動する関数の分類

exec 関数群を利用すると，親プロセスは終了してしまって，子プロセスが親プロセスの代わりに起動されます．これは UNIX の exec 関数に似た機能を実現したものですが，spawn 関数群でも同じことができるので，利用することは少ないでしょう．

＊3 MS-C 以外の処理系でもほぼ同じ関数が提供されているが，関数名や仕様の異なるものは，**表 4-7** と**表 4-8** の脚注として示している

<spawn(P_WAIT)>

親プロセスは子プロセスが終了するまで待つ

<exec, spawn(P_OVERLAY)>

起動
親プロセス
終了
起動
spawn
exec
子プロセス
終了

親プロセスが終了して子プロセスが起動する

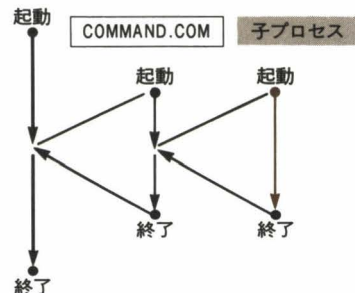
<system>

COMMAND.COMを起動したあとに，子プロセスを起動する

図4-7　子プロセスを起動する関数群の違い

関数名	書式	返値	各引数のタイプ
spawnv() execv()	int spawnv(mode, path, arg) ; int execv(path, arg) ;	エラーコード，もしくは子プロセスの終了コード(spawn関数群のみ)[†2]	int mode ; char *path ; char *arg[ ] ; char *arg0, *arg1, …, *argn ; char *envp[ ] ;
spawnvp() execvp()	int spawnvp(mode, path, arg) ; int execvp(path, arg) ;		
spawnve()[†1] execve()	int spawnve(mode, path, arg, envp) ; int execve(path, arg, envp) ;		
spawnl() execl()	int spawnl(mode, path, arg0, arg1, …, argn, NULL) ; int execl(path, arg0, arg1, …, argn, NULL) ;		
spawnlp() execlp()	int spawnlp(mode, path, arg0, arg1, …, argn, NULL) ; int execlp(path, arg0, arg1, …, argn, NULL) ;		
spawnle()[†1] execle()	int spawnle(mode, path, arg0, arg1, …, argn, NULL, envp) ; int execle(path, arg0, arg1, …, argn, NULL, envp) ;		

Lattice Cでは，spawn関数群のかわりにfolk関数群を用いる
†1 ：LSI C-86にはない
†2 ：Lattice Cでは終了コードをwait関数で得る

表4-7　spawn関数群とexec関数群

spawn 関数群を利用した場合，親プロセスはそのままの状態で子プロセスの終了を待ち続けることもできます．これは，関数呼び出しによく似たものと考えればわかりやすいでしょう．

system 関数は，spawn 関数群の関数と同様に，親プロセスは子プロセスの終了を待つのですが，常に COMMAND.COM を起動します．system 関数の使い方については次の項で説明します．

spawn 関数群と exec 関数群にはそれぞれ 6 個の関数がありますが，exec 関数群は spawn 関数群のサブセットですから，2 つの関数群は同じものと考えることができます．表 4-7 に 6 種類の関数の仕様を示します．

spawn 関数群の関数（以下 spawn 関数と呼ぶ）は，exec 関数群の機能を完全に持っていますから，以降の説明は spawn 関数についてのみ行います．

すべての関数の第 1 引数 mode には，次のマクロのいずれかを指定します．この引数によって，親プロセスが子プロセスの実行終了を待つか，子プロセスの実行前に終了するか（子プロセスと入れ換わるか）を指定するわけです．なお，これらのマクロは，インクルードファイル「PROCESS.H」中で定義されています．

P_WAIT　　…子プロセスが終了するまで親プロセスは待つ
P_OVERLAY…親プロセスを終了させてから，子プロセスを実行する（exec関数群と同じ）

起動するコマンド名は第 2 引数として指定します．このコマンド名には拡張子を指定しなくても，spawn 関数は自動的に「.com」と「.exe」を付加してファイルを検索します．ただし，spawnlp 関数と spawnvp 関数以外の spawn 関数は，環境変数 PATH を参照しませんから注意してください．また，コマンド行からコマンドを起動する場合，通常はバッチファイルも自動的に検索されますが，spawn 関数はバッチファイルを無視してしまいます．また，spawn 関数は実行ファイルであるかどうかを調べずにメモリ上にロードして，実行してしまうので，実行ファイル以外のファイルをコマンド名として指定すると暴走します．

子プロセスに渡す引数の指定方法は関数によって異なり，2 つの方法があります．spawnl 関数，spawnlp 関数，spawnle 関数では，子プロセスへの引数をそのまま関数の引数として並べ，その後ろに NULL を指定します．たとえば，「WC　A.C」というコマンドを実行する場合には，

というように指定します．

spawnv関数，spawnvp関数，spawnve関数の場合には，コマンドへの引数はポインタ配列の形で指定します．ポインタ配列の最後の要素にはNULLを格納してデータの終わりを示します．たとえば，上と同様に「WC A.C」を実行したい場合には，

```
char *arg[3];
arg[0] = "WC";
arg[1] = "A.C";
arg[2] =NULL;
spawnvp(P_WAIT, "WC", arg);
```

というようにして実行します．

引数の渡し方で注意してほしいのは，どちらの方法をとる場合でも，最初の要素としてコマンドの名前を指定する必要があることです．実は，現在のMS-DOSでは，このコマンド名はダミーであり，子プロセスに渡されることはないのですが，このコマンド名の指定を忘れると，最初の引数がコマンドに渡されないことになってしまいます．もう1つ注意する必要があるのは，どちらの場合も，かならず引数の最後にNULLを指定することです．このNULLの指定を忘れると，spawn関数は引数の終わりを見つけられずに，余分な引数を渡してしまったり，エラーで返ってくることになります．

spawnve関数とspawnle関数では，最後の引数として子プロセスの環境（環境変数の集まり）を指定します．この2つ以外のspawn関数は，自分の環境をそのまま子プロセスに渡すのですが，これらの関数を利用することによって，自分とは異なる環境を子プロセスに渡すことができます．環境は，ポインタ配列の形で指定します．

まとめとして，表4-8にspawn関数の機能や仕様による早見表を示しておきましょう．

子プロセスの引数の渡し方		環境変数PATHによるコマンドの自動検索	子プロセスの環境の指定	子プロセスの環境
関数の引数	ポインタ配列			
spawnl( )	spawnv( )	な　し	な　し	親プロセスと同じ
spawnlp( )	spawnvp( )	あ　り	な　し	親プロセスと同じ
spawnle( )	spawnve( )	な　し	あ　り	引数として指定されたもの

注：Turbo CとLattice CにはPATHの検索と子プロセスの環境の指定がともに可能なswawnlpe関数（folklpe関数）が用意されている

表4-8　spawn関数の機能

■ system関数の使い方

spawn関数を利用すれば，MS-DOSの実行ファイルはすべて子プロセスとして実行できますが，実際に子プロセスを起動する場合，次のような点で問題が起きてしまいます．

・COMMAND.COMの内部コマンドが直接実行できない

DIRやCDコマンドのようなCOMMAND.COMの内部コマンドは，COMMAND.COM自身を実行ファイルとする必要がある．

・バッチファイルが実行できない

バッチファイルはCOMMAND.COMが処理するファイルであり，MS-DOSの実行ファイルではないため，spawn関数では起動できない．

・リダイレクトの処理が面倒である

コマンド行で「>」，「<」を利用して簡単に実現できるリダイレクトが，spawn関数を利用する場合はコマンドラインをそのまま記述することができないため，非常に面倒になってしまう．

上に挙げた問題点を解決するためにはsystem関数を利用します．system関数は，指定された文字列を引数としてCOMMAND.COMを起動するため，コマンド行と同じように，さまざまな機能を利用することができます[*4]．たとえば，カレントドライブをCドライブに変更する場合には，前節のintdos関数を利用しても実現できますが，system関数を利用すれば，次のように1行で記述が可能です．

```
system("C:");
        └──── コマンドラインと同じように記述できる
```

一方，system関数を利用する場合のデメリットとしては，次のような点を挙げることができます．

- 直接コマンドを起動する場合に比べて，COMMAND.COMの起動などに時間がかかるため，実行速度が遅くなる．
- COMMAND.COMが常に0で戻ってくるため，コマンドのリターンコードを親プロセスが知ることができない．

実行時間が遅くなることは，いうまでもないことでしょう．直接子プロセスを起動すれば実行ファイルの読み込みは1回ですみますが，COMMAND.COMを介してコマンドを実行する場合には2回のファイル読み込みが必要になります．COMMAND.COM自身も比較的大規模なコマンドですから，起動時の初期化などに時間がかかるのは仕方のないことです．

＊4　MS-C以外の処理系のなかには，system関数がCOMMAND.COMを起動しないため，実行ファイルしか起動できないものもある．

2番目の問題はCOMMAND.COM，言い換えるとMS-DOS特有の問題です．UNIXのシェル(MS-DOSのCOMMAND.COMに相当する)では，シェルから起動したプログラムのリターンコードをきちんと管理しており，そのエラーコードによってシェル自身のエラーコードが変化します．そのため，シェルを介して子プロセスを起動しても，処理に時間がかかるだけで，エラーが起きたかどうかを親プロセスは知ることができます．それに対してCOMMAND.COMは，起動した子プロセスのリターンコードに関係なく，常にリターンコードとして0を返してしまいます．そのため，親プロセス側では，処理が正常に終了したのかどうかを知るすべがありません．

最後に，この節のまとめとしてspawn関数のサンプルプログラムをかねて，指定されたコマンドを起動するための関数，execute関数を**リスト4-9～リスト4-11**に紹介します．execute関数は，カレントディレクトリと環境変数PATHで指定されたディレクトリを検索して，拡張子「.COM」または「.EXE」のファイルを見つけるとそのファイルを実行します．コマンドが見つからなかった場合にはCOMMAND.COMが起動され，引数としてコマンドが渡されます(内部コマンドやバッチファイルを実行する場合)．各コマンドの実行後，そのリターンコードを表示します．

```
 1: /*
 2:  *   execute.c: command execution library.
 3:  *
 4:  *   compile option:
 5:  *       -DKANJI recognize shift jis in command name
 6:  */
 7:
 8: #include <stdio.h>
 9: #include <stdlib.h>
10: #include <string.h>
11: #include <ctype.h>
12: #include <process.h>
13:
14:
15: /*
16:  *   execute(): execute a command.
17:  *   ret = exec_cmd(line);
18:  *   int      ret;          0 if success.
19:  *   char     *line;        line to execute.
20:  */
21:
22: int execute(char *line)
23: {
24:     char     cmd[256];
25:     char     *p, *q;
26:     int      status;
27:     char     *shell;
28:
```

```
29:     fflush(stderr);  } バッファのフラッシュ
30:     fflush(stdout);  }              ──→環境を検索する関数
31:     if ((shell = getenv("COMSPEC")) == NULL) } コマンドプロセッサ
32:         shell = "command";                   } のパスの決定
33:     for (p = cmd, q = line; *q != '¥0' && !isspace(*q); ) { …コマンド名の
34: #ifdef KANJI    └→コマンド名  └→コマンドの引数                切り出し
35:         if (iskanji(q[0]) && iskanji2(q[1])) {
36:             *p++ = *q++;
37:             *p++ = *q++;
38:         } else
39: #endif
40:             *p++ = *q++;
41:     }
42:     *p = '¥0';                    直接実行できなければ，COMMAND.COMを起動
43:     if ((status = spawnlp(P_WAIT, cmd, cmd, q, NULL)) == -1)
44:         status = spawnlp(P_WAIT, shell, shell, "/c", line, NULL);
45:     return status;
46: }
47:
48: /* ---- end of execute.c ---- */
```

リスト 4-9　EXECUTE.C

```
 1: /*
 2:  *  exectest.c: test program for execute()
 3:  */
 4:
 5: #include <stdio.h>
 6: #include <stdlib.h>
 7: #include <string.h>
 8:
 9: main()……………………execute関数用テストプログラム
10: {                    標準入力から１行ずつ読み込み，実行してその返値を表示する
11:     char    buf[256];
12:     int     i;
13:     char    *p;
14:     char    *strchr();
15:
16:     while (fgets(buf, sizeof(buf), stdin) != NULL) {
17:         if ((p = strchr(buf, '¥n')) == NULL)
18:             exit(1);
19:         *p = '¥0';
20:         i = execute(buf);
21:         printf("return value = %d¥n", i);
22:     }
23:     exit(0);
24: }
25:
26: /* ---- end of exectest.c ---- */
```

[実行結果]

```
B>exectest
ld exectest.exe
00100000    10705 10/01/90 09:18:32 EXECTEST.EXE
return value = 0
ld z:foo
ld: z:foo: No such file or directory.
return value = 2
dir z:foo
ドライブの指定が違います.
return value = 0
^Z

B>
```

リスト4-10　EXECTEST.C

```
1: execute.obj: execute.c
2:      cl -AS -c execute.c
3:
4: exectest.obj: exectest.c
5:      cl -AS -c exectest.c
6:
7: exectest.exe: exectest.obj execute.obj
8:      cl -Feexectest.exe exectest.obj execute.obj
```

リスト4-11　MAKEFILE

4.4 サンプルプログラム —FINDF—

本章のまとめとして，MS-DOS 上に UNIX の find コマンドのサブセット FINDF を作成します．実用に耐えるコマンドとするために，前節まででは説明しなかった MS-DOS のファンクションコールなども利用しています．プログラムをじっくり読んで，プログラム開発の参考にしてください．なお，このプログラムは MS-C で記述され，MS-DOS Ver3.1/3.30 でテストしました．

■ FINDFコマンドの概要

FINDF コマンドは，MS-DOS の階層化ディレクトリを検索し，特定のファイルに対して処理を行うためのコマンドです．コマンド行でパスと式を指定すると，指定されたパスから下のディレクトリをすべて検索し，ファイル名やファイルサイズ，タイムスタンプなど，さまざまな条件でファイルを選びだし，指定した処理(ファイル名の表示や別コマンドの実行など)を行うことができます．とくにハードディスクを使って多くのファイルを階層ディレクトリで管理している場合は，その保守などに役立つでしょう．

コマンド行の書式

FINDF コマンドの書式は，次のようになっています．

FINDF ＜スタートパス＞… ＜式＞

スタートパスは，どのパスから検索を開始するかを指示します．複数のパスを書くと，最初のスタートパスから順に検索します．式は，次に説明する「項」の並んだものです．FINDF は，検索中にディレクトリエントリが見つかるごとにこの式を評価します．

FINDF の項

項は，FINDF でファイルの検索条件や，そのファイルを処理するためのコマンドを記述するものです．式中の項は左から順に評価され，いずれかの項の値が「偽」になるかどうかを式の最後まで評価し，次のエントリの検索に移ります．FINDF には，次のような項が用意されています．

-attr <list>

<list>で指定したすべての属性が，処理対象のファイルの属性としてついていた場合にのみ「真」となります．<list>で指定できる文字は次ページの**表 4-9** とおりです．

文字	意　味
o	書き込み禁止属性
h	不可視属性
s	システムファイル
d	ディレクトリ属性
a	保存ビット

表4-9　attrで指定できる文字(ディレクトリの属性)

-exec <コマンド> <引数> ;

<コマンド>で指定されたコマンドを<引数>をつけて実行し，その実行結果(実行したコマンドのリターンコード)が0の場合のみ「真」となります．<引数>のうち前後を空白で囲まれた「{}」という文字列は，現在処理中のファイル名(パスつき)に置換されます．また，引数の最後を表す「；」と引数のあいだは空白で区切る必要があります．

-name <パターン>

現在処理しているファイル名と<パターン>を比較し，マッチした場合のみ「真」となります．<パターン>として指定可能なワイルドカードは，一般的なMS-DOS風のものではなく，**表4-10**に示すUNIX風のワイルドカードになります．この仕様ですべてのファイル名と一致させる指定は，「***.***」ではなく「*****」としなければなりません．

特殊文字	意　味
*	0文字以上の任意の文字
?	任意の1文字
[ ]	囲まれた文字のいずれか．〝-〟を使って連続した文字を表すことも可能

表4-10　FINDFコマンドのワイルドカード

-newer <ファイル名>

現在処理しているファイルと，<ファイル名>で指定されたファイルのタイムスタンプを比較して，処理中のファイルの方が新しい場合に「真」となります．

-ok <コマンド> <引数> ;

基本的な機能は「-exec」とまったく同じですが，実行前にコマンド名と現在処理中のファイル名を表示し，ユーザーの確認を求めます．「Y」もしくは「y」を入力した場合のみコマンドは実行されます．

-print

現在処理中のファイル名を標準出力に出力します．項の値は常に「真」となります．

-size{<n>|+<n>|−<n>}

ファイルのブロックサイズ(1ブロックは512バイト)を調べ，条件を満たした場合のみ「真」となります．<n>と指定するとファイルのブロックサイズが<n>のとき，+<n>を指定すると<n>以上のとき，−<n>と指定すると<n>以下のときに，それぞれ「真」となります．

-time{<n>|+<n>|−<n>}

ファイルのタイムスタンプと現在時刻を比較し，条件を満たした場合のみ「真」となります．<n>と指定すると<n>日前のもの，+<n>と指定すると<n>日前より前のもの，−<n>と指定すると<n>日前より後に作られたものに対して「真」となります．

-type <属性>

現在処理中のファイルが，ファイルかディレクトリかを調べ，条件を満たした場合のみ「真」になります．属性としては**表4-11**のいずれかを指定できます．

文字	意　味
d	ディレクトリの場合のみ真
f	一般ファイルの場合のみ真

表4-11　typeで指定できる文字

演算子

FINDFの式中では，前述の項とともに**表4-12**に示す演算子を利用して，より複雑な条件を表現することができます．FINDFの式は，3つの演算子の指定が可能ですが，項を並べるだけで論理積(AND条件)で結合されたことになるので，合計4種類の演算子が利用できることになります．複数の演算子を組み合わせるときは，その結合順序に注意してください．

結合順序	演算子†	意味	機　　能
高 ↑	( )	かっこ	かっこ内に記述された項を優先的に結合する
	!	NOT	直後の項の否定をとる
	なし	AND	デフォルトの結合規則．前後の項の論理積をとる
↓ 低	-o	OR	前後の項の論理和をとる

† 各演算子の前後には空白が必要

表4-12　演算子と結合順序

実行例

FINDFコマンドは，条件の指定が非常に複雑なため，これまでの説明では少々わかりにくいかと思います．そこで以下に記述例を示しますので，コマンドの仕様を理解する一助としてください(図4-8)．

```
findf ¥ -print
findf ¥ -name *.bak -ok del {} ;
findf . -type f -size +200 -print
findf . -type f -name *.[ch] -print -exec type {} > srcs ;
findf ¥ ( -name #* -o -name core ) -type f -time +7 -exec del {} ;
```


図4-8　FINDFコマンドの実行例

■ プログラムの内部構造

FINDFコマンドは大きく分けて，MS-DOSとのインターフェイスをつかさどる部分と，コマンド行で指定された式を内部構造に変換して評価する部分との2つに分けることができます．MS-DOSとのインターフェイスについては，すでに前項で説明したので，ここではプログラムの全体構造と，式の解釈・評価をどのように実現しているかについて説明します．

プログラムの構造

FINDFコマンドの各モジュールは，次のような呼び出し関係をもっています．MAIN.Cモジュールは，基本的にはほかのモジュールを呼び出したり，エラーメッセージを表示するだけのモジュールになっており，EXP.CとEXEC.Cがコマンドの機能を実現しています．DIRLIB.C，DOCMD.C，STR.C，TIME.CはEXP.CとEXEC.Cの下請けルーチンの集まりです．

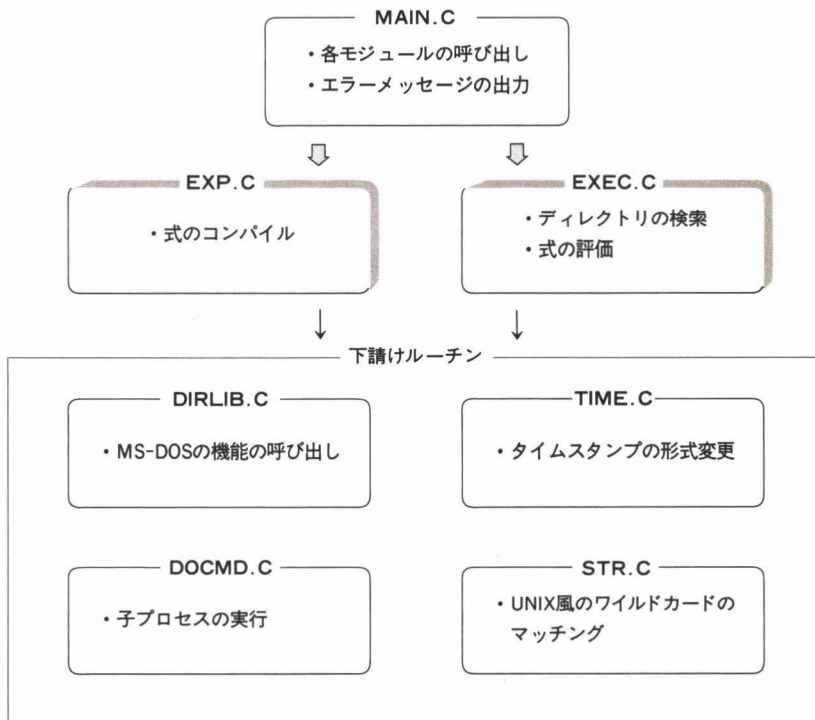



図 4-9　モジュールの構造

式の内部構造

コマンド行で指定された式は，最初に式の解釈ルーチン(EXP.C モジュール)によって，FINDF の内部構造に変換されます．内部構造では，式の項や演算子を 1 つのノードとする「木構造」として式が表現されます．ノードの構造は DEFS.H 中で定義されています．例として，次のような式の内部構造のノードによる表現を次ページの図 4-10 に示します．

```
-attr a ( ! -name *.exe -o -size -20 ) -print
```

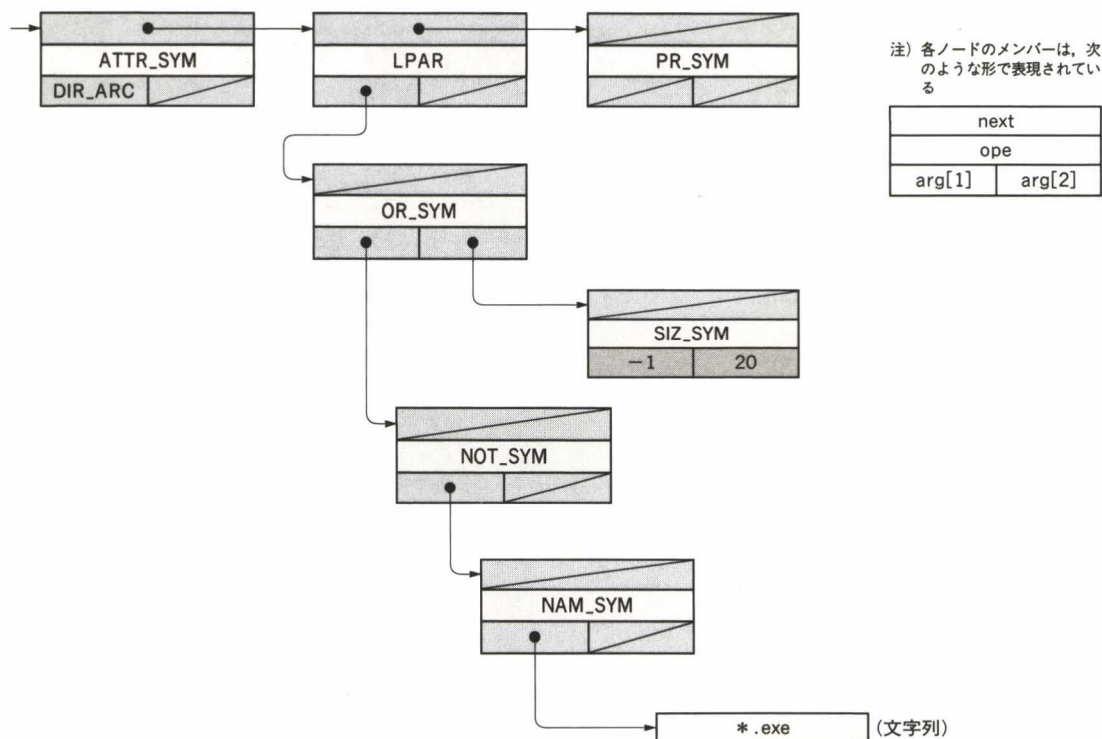



図4-10　ノードの定義

パスの検索と式の評価

コマンド行で与えられた式を内部構造に変換できた(エラーがなかった)場合には，EXEC.C モジュールがパスを検索しながら式の評価を行います．パスの検索ルーチン(srch_rec 関数)の構造は，基本的には LD コマンドと同様になっていますが，サブディレクトリを検索するために，自分自身を再帰的に起動しながら処理を行っています．

MS-DOS インターフェイス

intdos 関数によって，MS-DOS のファンクションコールを直接起動する関数は，すべて DIRLIB.C 中にまとめてあります．これはリスト 4-2 で示した DOSFIND.C をさらに拡張した仕様になっています．ここで定義されている関数は，ほとんどが基本的なものばかりですから，ほかのプログラムでもそのまま利用できるものが多いはずです．

文字列処理

STR.C モジュールでは,「-name」の処理を行うために,ワイルドカードのマッチングルーチンなどを定義しています.UNIX 風のワイルドカードの実現と日本語処理を行うために少々複雑に見えますが,注意深くプログラムを追えば読みこなすことは難しくないでしょう.

時間情報の変換

TIME.C モジュールは,MS-DOS のタイムスタンプの表現を UNIX 風のタイムスタンプ(1970 年 1 月 1 日 0 時 0 分 0 秒からの通算秒)に変換するための関数を定義しています.閏年の処理などを行うために,1968 年 3 月 1 日を基点に処理を行っていることに注意してください.

子プロセスの実行

DOCMD.C モジュールは,「-exec」と「-ok」を実現するために,子プロセスを実行するモジュールです.基本的な構造は,子プロセスの実行の節で紹介した execute 関数とほぼ同等のものですが,FINDF の内部構造にあわせて,引数の形式に変更を加えて,引数の置換を行うようにしています.

プログラムの移植

FINDF は,MS-DOS に依存したプログラムの一例として,MS-DOS の DIR 構造体をプログラム全体で利用していますので,ほかの OS に移植することは難しいでしょう.MS-DOS 上のほかの処理系に移植する場合には,intdos 関数を利用している DIRLIB.C モジュールと spawn 関数を利用している DOCMD.C モジュールを中心に,エラーチェックなどを書き直してください.

```
 1: DEST     =    .
 2: HDRS     =    btype.h defs.h dir.h misc.h
 3: CFLAGS   =
 4: LDFLAGS  =    -link /stack:8192
 5: LIBS     =
 6: MEM      =    S
 7: LINKER   =    $(CC)
 8: MAKEFILE=     makefile
 9: OBJS     =    dirlib.obj docmd.obj exec.obj exp.obj main.obj str.obj
    time.obj
10: PRINT    =    jpr
11: PROGRAM  =    findf.exe
12: SRCS     =    dirlib.c docmd.c exec.c exp.c main.c str.c time.c
13:
14: all : $(PROGRAM)
15:
16: $(PROGRAM) : $(OBJS) $(LIBS)
17:      $(LINKER) $(CFLAGS) $(OBJS) $(LIBS) -o $* $(LDFLAGS)
```

```
18:
19: clean :
20:       rm -f $(OBJS)
21:
22: print : $(HDRS) $(SRCS)
23:       @$(PRINT) $?
24:       @touch $@
25:
26: program : $(PROGRAM)
27:
28: ###
29: dirlib.obj :  btype.h misc.h dir.h
30: docmd.obj : btype.h misc.h
31: exec.obj : btype.h misc.h dir.h defs.h
32: exp.obj : btype.h misc.h dir.h defs.h
33: main.obj : btype.h misc.h
34: str.obj : btype.h misc.h
35: time.obj : btype.h misc.h
```

リスト4-12　MAKEFILE(UNIXコンパチブルなMAKE用)

```
 1: #Makefile of findf for MS-MAKE ver.3
 2:
 3: str.obj: btype.h misc.h str.c
 4:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c str.c
 5:
 6: time.obj: btype.h misc.h time.c
 7:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c time.c
 8:
 9: dirlib.obj: btype.h misc.h dir.h dirlib.c
10:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c dirlib.c
11:
12: docmd.obj: btype.h misc.h docmd.c
13:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c docmd.c
14:
15: exec.obj: btype.h misc.h defs.h dir.h exec.c
16:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c exec.c
17:
18: exp.obj: btype.h misc.h defs.h dir.h exp.c
19:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c exp.c
20:
21: main.obj: btype.h misc.h main.c
22:     cl -DLINT_ARGS -DNO_EXT_KEYS -AS -c main.c
23:
24: findf.exe: main.obj exp.obj exec.obj docmd.obj dirlib.obj time.obj
    str.obj
25:     cl -o findf main exp exec docmd dirlib time str -link /stack:8192
```

リスト4-13　MAKEFILE.MS(MS-DOS Ver3.1以降付属のMAKE用)

```
 1: /*
 2:  *  btype.h: basic type definition header
 3:  */
 4:
 5: #ifndef NULL ……………………NULLの定義
 6: #if (!defined(M_I86CM) && !defined(M_I86LM))
 7: #define NULL    0
 8: #else
 9: #define NULL    0L
10: #endif
11: #endif
12:
13: #define TRUE                1……………………「真」の値
14: #define FALSE               0……………………「偽」の値
15: #define SUCCESS             TRUE ……………「成功」の値
16: #define FAIL                FALSE……………「失敗」の値
17:
18: typedef int                 BOOL;……………論理型の型定義
19: typedef unsigned char       uchar;  ┐
20: typedef unsigned int        uint;   │
21: typedef unsigned short      ushort; │ プログラムで利用しているいろいろな
22: typedef unsigned long       ulong;  │ 型の省略記法の定義
23: typedef unsigned short      kchar;  ┘
24:
25: /* ---- end of btype.h ---- */
```

リスト 4-14　BTYPE.H

```
    /*
 _.  *  defs.h: definition for find command ……………………式の内部表現に使う型や
 3:  */                                                     構造体の定義
 4:
 5: typedef enum { ……………………項と演算子を表すシンボル
 6:     OR_SYM, NOT_SYM, EXE_SYM, OK_SYM, NAM_SYM,
 7:     NEW_SYM, PR_SYM, SIZ_SYM, TIM_SYM, TYP_SYM,
 8:     ATTR_SYM, LPAR, RPAR, SEMI, ILL_SYM
 9: } ope_t;
10:
11: typedef struct   _node node_t;……………………ノードの型宣言
12: typedef union    _arg arg_t;………………………ノード中の引数の型宣言
13:
14: union _arg { ………………各ノードに含まれる引数の格納される共用体
15:     node_t  *nodep;
16:     char    *symbol;
17:     char    **arglist;
18:     long    value;
19: };
20:
21: struct  _node { ……………………ノードの型定義
22:     node_t  *next;……………次のノードへのポインタ
23:     ope_t   ope;……………………項や演算子の種別
24:     arg_t   arg[2];……………引数
25: };
26:
27: /* ---- end of defs.h ---- */
```

リスト 4-15　DEFS.H

```
 1: /*
 2:  *  dir.h: Definition for MS-DOS directory entry structure.
 3:  */
 4:
 5: /* ------- bit definitions of attribute byte ------------ */
 6:
 7: #define DIR_RO  0x0001          /* read only        */
 8: #define DIR_HID 0x0002          /* hidden file      */
 9: #define DIR_SYS 0x0004          /* system file      */
10: #define DIR_VOL 0x0008          /* volume name      */
11: #define DIR_DIR 0x0010          /* directory entry  */
12: #define DIR_ARC 0x0020          /* archive bit      */
13:
14: /* ------- structure definitions of DIR structure ------ */
15:
16: typedef struct {
17:     char    _reserved[21];      /* reserved area    */
18:     uchar   _attrib;            /* attribute byte   */
19:     ushort  _ftime;             /* time of file     */
20:     ushort  _fdate;             /* date of file     */
21:     ulong   _fsize;             /* file size        */
22:     uchar   d_name[13];         /* packed file name */
23: } DIR;
24:
25: /* ---- end of dir.h ---- */
```

リスト4-16　DIR.H

```
 1: /*
 2:  *  misc.h: miscellaneous definition for find command……FINDFプログラム用の
 3:  */                                                     雑多な宣言
 4:
 5: #include <assert.h>
 6:
 7: #define xalloc(type,num)     ((type *)malloc(sizeof(type) * (num)))
 8: #define strequ(a,b)          (strcmp((a),(b))==0)
 9: #define strtail(a)           ((a)+strlen(a))
10:                                                                  マクロ
11: /* -------- return value definition of filetype() -------- */
12:
13: #define PATH_ERROR        (-2)     /* non exist drive */
14: #define PATH_AMBIGUOUS    (-1)     /* contain wild-card */
15: #define PATH_NONE         (0)      /* unexist path. */      filetype関数の
16: #define PATH_ROOTDIR      (1)      /* root directory. */    返値の定義
17: #define PATH_SUBDIR       (2)      /* sub directory. */
18: #define PATH_REGULAR      (3)      /* normal file. */
19:
20: /* ---- end of misc.h ---- */
```

リスト4-17　MISC.H

```
 1: /*
 2:  *  main.c : main module for find command ……………メインモジュール
 3:  */
 4:
 5: #include <sys/types.h>
 6: #include <stdio.h>
 7: #include <ctype.h>
 8: #include <stdlib.h>
 9: #include <string.h>
10: #include "btype.h"
11: #include "misc.h"
12:
13: time_t  current_time;……………findfコマンドの起動時刻
14:
15: void    message(char *s1, char *s2, char *s3, char *s4, char *s5,
    char *s6)        └→メッセージの出力
16: {
17:     fprintf(stderr, "findf: ");
18:     fprintf(stderr, s1, s2, s3, s4, s5, s6);
19: }
20:             ┌→回復不可能なエラー処理
21: void    fatal(char *s1, char *s2, char *s3, char *s4, char *s5,
    char *s6)
22: {
23:     fprintf(stderr, "findf: ");
24:     fprintf(stderr, s1, s2, s3, s4, s5, s6);
25:     exit(2);
26: }
27:
28: void    usage()………………コマンド行の書式表示
29: {
30:     fprintf(stderr, "Usage: findf <startpath>... <expression>¥n");
31:     exit(2);
32: }
33:
34: void    main(int argc, char **argv)
35: {
36:     int     ndir;
37:     char    **pdir;
38:
39:     if (argc == 1)
40:         usage();
41:     argv++;
42:     pdir = argv;
43:     for (ndir = 0; --argc > 0; argv++) {…………スタートパスのスキップとカウント
44:         if (**argv == '-' || strequ(*argv, "(") || strequ(*argv,"!"))
45:             break;
46:         ndir++;
47:     }
48:
49:     if (ndir == 0)
50:         fatal("Path not specified.");
51:
```

MS-Cなどでは，「varargs.h」を利用することによって，よりスマートに記述できる（15～26行目）

```
52:     if (argc == 0)
53:         fatal("Expression list not specified.");
54:     parse_exp(argc, argv); ………………式のコンパイル
55:
56:     gettime(&current_time); ………………現在時刻の取得
57:     while (ndir-- > 0)  } パスの検索と式の実行
58:         find(*pdir++);  }
59:     exit(0);
60: }
61:
62: /* ---- end of main.c ---- */
```

リスト 4-18　MAIN.C

```
 1: /*
 2:  *  exp.c : module for parse expression list of find command ………… 式をコンパイルし, ノードを生成するルーチン
 3:  */
 4:
 5: #include <sys/types.h>
 6: #include <stdio.h>
 7: #include <ctype.h>
 8: #include <stdlib.h>
 9: #include <string.h>
10: #include <jstring.h>
11: #include "btype.h"
12: #include "misc.h"
13: #include "dir.h"
14: #include "defs.h"
15:
16: extern  time_t  to_timet();
17:
18: static  int     argcnt;
19: static  char    **argptr;
20: static  node_t  *or_exp();
21: node_t  *exp_list = NULL;
22:
23: static  void    incomplete() …………引数不足エラーの処理ルーチン
24: {
25:     fatal("Incomplete statement.");
26: }
27:
28: static  void    alloc_error() ……………メモリ不足エラーの処理ルーチン
29: {
30:     fatal("Not enough core. Stop execution.");
31: }
32:
33: static  node_t  *mknode(ope_t ope) …………新しいノードを生成するルーチン
                            └──→ノードの型
34: {
35:     node_t  *p;
36:
```

```
37:     if ((p = xalloc(node_t, 1)) == NULL)
38:         alloc_error();
39:     p->ope = ope;
40:     p->next = NULL;
41:     p->arg[0].value = 0L;
42:     p->arg[1].value = 0L;
43:     return p;
44: }
45:
46: static  ope_t   srch_ope(char *s)………………コマンド行の項・演算子を,
47: {                                          内部の型に変換するルーチン
48:     struct _symtbl {
49:         char    *sym;
50:         ope_t   type;
51:     };
52:     struct  _symtbl *p;
53:     static  struct  _symtbl symtbl[] = {
54:         {"-o",      OR_SYM},
55:         {"!",       NOT_SYM},
56:         {"-exec",   EXE_SYM},
57:         {"-ok",     OK_SYM},
58:         {"-name",   NAM_SYM},
59:         {"-newer",  NEW_SYM},
60:         {"-print",  PR_SYM},
61:         {"-size",   SIZ_SYM},
62:         {"-time",   TIM_SYM},
63:         {"-type",   TYP_SYM},
64:         {"-attr",   ATTR_SYM},
65:         {"(",       LPAR},
66:         {")",       RPAR},
67:         {";",       SEMI},
68:         {NULL,      ILL_SYM}………………表の終わりを表すエントリ
69:     };
70:
71:     for (p = symtbl; p->sym != NULL; p++)  ┐
72:         if (strequ(p->sym, s))             ├リニアサーチ
73:             return p->type;                ┘
74:     return ILL_SYM;………………未定義シンボルのときの返値
75: }
76:
77: static  char    *next_arg()………………次の引数を得るルーチン
78: {
79:     if (argcnt <= 0)
80:         return NULL;
81:     argcnt--;
82:     return *argptr++;
83: }
84:
85: static  void    back_arg()………………next arg関数のキャンセル・ルーチン
86: {                                   (getc関数に対するungetc関数と同じ)
87:     argcnt++;
88:     argptr--;
89: }
90:
```

```
 91: static  node_t  *primary_exp()……………基本的な項のコンパイル・ルーチン
 92: {
 93:     node_t  *p;
 94:     char    *cp;
 95:     ope_t   ope;
 96:     int     i, j;
 97:     long    atol();
 98:     DIR     dtabuf;
 99:
100:     if ((cp = next_arg()) == NULL)
101:         incomplete();
102:     switch (ope = srch_ope(cp)) {……………項／演算子別の処理
103:     case LPAR:
104:         p = mknode(ope);
105:         p->arg[0].nodep = or_exp();……………再帰的にコンパイル・ルーチンを起動
106:         if ((cp = next_arg()) == NULL)
107:             incomplete();
108:         if (srch_ope(cp) != RPAR)……………")"で終わっているかどうかをチェック
109:             fatal("Bad option <%s>.", cp);
110:         break;
111:
112:     case EXE_SYM:  ⎫
113:     case OK_SYM:   ⎭"-exec"と"-ok"のコンパイルの仕方はまったく同じ
114:         p = mknode(ope);
115:         p->arg[0].arglist = argptr;……main関数の引数argvへのポインタをそのまま格納
116:         i = 0;
117:         while ((cp = next_arg()) != NULL && srch_ope(cp) != SEMI) ⎫
118:             i++;                                                  ⎬……コマンド行の引数の数をカウント
119:         if (cp == NULL)                                           ⎪
120:             incomplete();                                         ⎭
121:         p->arg[1].value = i;
122:         break;
123:
124:     case NAM_SYM:
125:         if ((cp = next_arg()) == NULL)……………パターン("-name"の引数)の取得
126:             incomplete();
127:
128:         p = mknode(ope);
129:         jstrupr(p->arg[0].symbol = cp);……………MS-DOSでは，ファイル名は大文字と小文字を区別しない
130:         break;
131:
132:     case NEW_SYM:
133:         if ((cp = next_arg()) == NULL)………基準ファイル名("-newer"の引数)の取得
134:             incomplete();
135:         switch (filetype(cp)) {……filetype関数によって，基準ファイルの整合性をチェック
136:         case PATH_AMBIGUOUS:
137:             fatal("filename ambiguous <%s>.", cp);
138:         case PATH_NONE:
139:         case PATH_ERROR:
140:             fatal("Cannot access <%s>.", cp);
141:         case PATH_ROOTDIR:
142:             set_root_dta(&dtabuf);
143:             break;
144:         case PATH_SUBDIR:
145:         case PATH_REGULAR:
```

```
146:             if (!srchfirst(cp, &dtabuf))
147:                 assert(0); ……………………プログラムエラー(filetype関数のエラー)
148:             break;
149:         default:
150:             assert(0); ……………………
151:         }
152:         p = mknode(ope);
153:         p->arg[0].value = to_timet(dtabuf._fdate, dtabuf._ftime);
154:         break;
155: 
156:     case PR_SYM:
157:         p = mknode(ope);
158:         break;
159: 
160:     case SIZ_SYM:   } "-time", "-size"の引数につけられる符号(-, +)の処理
161:     case TIM_SYM:   }
162:         if ((cp = next_arg()) == NULL)
163:             incomplete();
164:         p = mknode(ope);
165:         switch (*cp) {
166:         case '+':
167:             p->arg[0].value = 1;
168:             i = 1;
169:             break;
170:         case '-':
171:             p->arg[0].value = -1;
172:             i = 1;
173:             break;
174:         default:
175:             p->arg[0].value = 0;
176:             i = 0;
177:             break;
178:         }
179:                                 /* check arg */
180:         for (j = i; cp[j] != '¥0'; j++)                      }
181:             if (!isdigit(cp[j]))                              } "-time", "-size"の
182:                 fatal("Illegal parameter <%s>.", cp);         } 引数の値の処理
183:         p->arg[1].value = atol(cp + i);                      }
184:         break;
185: 
186:     case TYP_SYM:
187:         if ((cp = next_arg()) == NULL)
188:             incomplete();
189:         if (strlen(cp) != 1)
190:             fatal("Illegal parameter <%s>.", cp);
191:         p = mknode(ope);
192:         switch (*cp) {
193:         case 'd':
194:             j = 0; ……………………………………exec.c中のchk_type関数で参照
195:             break;
196:         case 'f':
197:             j = 1; ……………………
198:             break;
199:         default:
200:             fatal("Unknown type <%c> for -type expression.", *cp);
201:         }
```

```
202:             p->arg[0].value = j;
203:             break;
204:
205:         case ATTR_SYM:
206:             if ((cp = next_arg()) == NULL)
207:                 incomplete();
208:             j = 0x00;
209:             for (i = 0; cp[i] != '¥0'; i++)
210:                 switch (cp[i]) {
211:                 case 'o':
212:                     j |= DIR_RO;
213:                     break;
214:                 case 's':
215:                     j |= DIR_SYS;
216:                     break;
217:                 case 'h':
218:                     j |= DIR_HID;
219:                     break;
220:                 case 'd':
221:                     j |= DIR_DIR;
222:                     break;
223:                 case 'a':
224:                     j |= DIR_ARC;
225:                     break;
226:                 default:
227:                     fatal("Unknown type <%c> for -attr expression.",
228:                         cp[i]);
229:                     break;
230:                 }
231:             p = mknode(ope);
232:             p->arg[0].value = j;
233:             break;
234:
235:         case ILL_SYM:
236:         default:
237:             fatal("Bad option <%s>.", cp);
238:         }
239:         return p;
240: }
241:
242: static  node_t  *unary_exp()          /* ! */
243: {
244:     node_t  *p;
245:     char    *cp;
246:
247:     if ((cp = next_arg()) != NULL && srch_ope(cp) == NOT_SYM) {
248:         p = mknode(NOT_SYM);
249:         p->arg[0].nodep = primary_exp();
250:     } else {
251:         back_arg();
252:         p = primary_exp();
253:     }
254:     return p;
255: }
256:
```

exec.c中のchk_attr関数で参照

"!"演算子の処理

```
257: static  node_t  *and_exp()       /* default operator */
258: {
259:     node_t  *topnode;
260:     node_t  *nodep;
261:     char    *cp;
262:     ope_t   ope;
263:
264:     topnode = nodep = unary_exp();
265:     while ((cp = next_arg()) != NULL) {
266:         if ((ope = srch_ope(cp)) != OR_SYM && ope != RPAR) {
267:             back_arg();
268:             nodep->next = unary_exp();
269:             nodep = nodep->next;
270:         } else {
271:             back_arg();
272:             break;
273:         }
274:     }
275:     return topnode;
276: }
277:
278: static  node_t  *or_exp()        /* -o */ ……………… "-o"演算子の処理
279: {
280:     node_t  *topnode;
281:     node_t  *nodep;
282:     char    *cp;
283:
284:     topnode = and_exp();
285:     while ((cp = next_arg()) != NULL) {
286:         if (srch_ope(cp) == OR_SYM) {
287:             nodep = mknode(OR_SYM);
288:             nodep->arg[0].nodep = topnode;
289:             nodep->arg[1].nodep = and_exp();
290:             topnode = nodep;
291:         } else {
292:             back_arg();
293:             break;
294:         }
295:     }
296:     return  topnode;
297: }
298:
299: void    parse_exp(int argc, char **argv) ……………… main関数から呼び出される
                                                         コンパイル・ルーチン
300: {
301:     argcnt = argc; }
302:     argptr = argv; } next_argで利用
303:
304:     exp_list = or_exp(); …………… 式をコンパイルしポインタをセット
305: }
306:
307: /* ---- end of exp.c ---- */
```

リスト 4-19 EXP.C

```
 1: /*
 2:  *   exec.c : execution interpreter for find ……ディレクトリの検索と，式の評価
 3:  */                                                 を行うモジュール
 4:
 5: #include <sys/types.h>
 6: #include <stdio.h>
 7: #include <ctype.h>
 8: #include <string.h>
 9: #include "btype.h"
10: #include "misc.h"
11: #include "dir.h"
12: #include "defs.h"
13:
14: #define SEC_PER_DAY (time_t)(24L*60L*60L) ……1日当たりの秒数
15: #define BYTE_PER_BLOCK   (512L) ……1ブロックのバイト数
16:
17: extern  node_t   *exp_list; ……式の先頭へのポインタ
18: extern  time_t   current_time; ……FINDFコマンドの起動時刻
19:
20: static  uchar    path[256];
21:
22: static  BOOL     chk_size(DIR *dta, long flag, long basesize) ……
23: {                                                     "-size"の評価ルーチン
24:     long     size;
25:     BOOL     result;
26:
27:     size = (dta->_fsize + BYTE_PER_BLOCK - 1) / BYTE_PER_BLOCK; ……
28:     switch ((int)flag) {                          ファイルのブロックサイズを求める
29:     case 1:
30:         result = size >= basesize;
31:         break;
32:     case 0:
33:         result = size == basesize;   サイズの比較
34:         break;
35:     case -1:
36:         result = size <= basesize;
37:         break;
38:     default:
39:         assert(0);   プログラムのエラー検出
40:     }
41:     return result;
42: }
43:
44: static  BOOL     chk_time(DIR *dta, long flag, time_t basetime) ……
45: {                                                     "-time"の評価ルーチン
46:     time_t   _time;
47:     BOOL     result;
48:     extern   time_t   to_timet();
49:
50:     _time = (current_time - to_timet(dta->_fdate, dta->_ftime))
51:         / SEC_PER_DAY; ……ファイルのタイムスタンプと，
                          現在時刻の差を求める
```

```
 52:     switch ((int)flag) {
 53:     case 1:
 54:         result = _time > basetime;
 55:         break;
 56:     case 0:
 57:         result = _time == basetime;   時間の比較
 58:         break;
 59:     case -1:
 60:         result = _time < basetime;
 61:         break;
 62:     default:
 63:         assert(0);
 64:     }
 65:     return result;
 66: }
 67:
 68: static  BOOL    chk_type(DIR *dta, long flag) ..........“-type”の評価ルーチン
 69: {
 70:     BOOL    result;
 71:
 72:     switch ((int)flag) {
 73:     case 0:     /* "-type d" */
 74:         result = (dta->_attrib & DIR_DIR) != 0;
 75:         break;
 76:     case 1:     /* "-type f" */
 77:         result = (dta->_attrib & DIR_DIR) == 0;
 78:         break;
 79:     default:
 80:         assert(0);
 81:     }
 82:     return result;
 83: }
 84:
 85: static  BOOL    chk_attr(DIR *dta, long attr) ..........“-attr”の評価ルーチン
 86: {
 87:     register uchar  attrib;
 88:
 89:     attrib = (uchar)attr;
 90:     return (dta->_attrib & attrib) == attrib;
 91: }
 92:
 93: static  BOOL    check_ok(char *cmd, char *path) ........“-ok”の確認用ルーチン
 94: {
 95:     uchar   buf[256];
 96:
 97:     fflush(stdout);
 98:     message("<%s ... %s> ? ", cmd, path);
 99:     fflush(stderr);
100:     if (fgets(buf, sizeof(buf), stdin) == NULL)
101:         return FALSE;
102:     return (*buf == 'y' || *buf == 'Y') ? TRUE : FALSE;
103: }
```

```
104:                         評価されるファイルのパス        式の実行ルーチン
105: static  BOOL    execlist(node_t *np, uchar *path, DIR *dta)
106: {                              評価する式          評価されるファイルの管理情報
107:     extern  time_t  to_timet();
108:
109:     for ( ; np != NULL; np = np->next)
110:         switch (np->ope) {
111:         case LPAR:
112:             if (!execlist(np->arg[0].nodep, path, dta)) ……自己再帰
113:                 goto fail;
114:             break;
115:         case OK_SYM:
116:             if (!check_ok(np->arg[0].arglist[0], path))
117:                 goto fail;
118:             /* fall through */ ……ユーザーからの許可を得たあとは，“-exec”の処理部へ制御が移る
119:         case EXE_SYM:
120:             if (!exec_cmd(np->arg[0].arglist,
121:                 (int)(np->arg[1].value), path))
122:                 goto fail;
123:             break;
124:         case NAM_SYM:
125:             if (!match_name(dta->d_name, np->arg[0].symbol))
126:                 goto fail;
127:             break;
128:         case NEW_SYM:
129:             if (to_timet(dta->_fdate, dta->_ftime)
130:                 <= np->arg[0].value)
131:                 goto fail;
132:             break;
133:         case PR_SYM:
134:             puts(path);
135:             break;
136:         case SIZ_SYM:
137:             if (!chk_size(dta, np->arg[0].value,
138:                 np->arg[1].value))
139:                 goto fail;
140:             break;
141:         case TIM_SYM:
142:             if (!chk_time(dta, np->arg[0].value,
143:                 np->arg[1].value))
144:                 goto fail;
145:             break;
146:         case TYP_SYM:
147:             if (!chk_type(dta, np->arg[0].value))
148:                 goto fail;
149:             break;
150:         case ATTR_SYM:
151:             if (!chk_attr(dta, np->arg[0].value))
152:                 goto fail;
153:             break;
154:         case NOT_SYM:
155:             if (execlist(np->arg[0].nodep, path, dta))
156:                 goto fail;
157:             break;
```

```
158:            case OR_SYM:
159:                if (!execlist(np->arg[0].nodep, path, dta)     …"-o"の左辺
160:                    !execlist(np->arg[1].nodep, path, dta))………"-o"の右辺
161:                    goto fail;                                  一方が
162:                break;                                          成功す
163:            case ILL_SYM:                                       れば真
164:            default:
165:                assert(0);
166:        }
167:    return TRUE;
168: fail:;
169:    return FALSE;
170: }
171:
172: static  void    srch_rec(DIR *dta)……指定されたエントリを評価し，ディレクトリであれ
173: {                                       ば再帰的に検索と処理を行う関数
174:     DIR     subdta;
175:     uchar   *p;
176:
177:     (void)execlist(exp_list, path, dta);……………式の実行，返値は無視
178:     if (!(dta->_attrib & DIR_DIR)) ……………………ディレクトリでなければリターン
179:         return;
180:     p = path;
181:     if (isalpha(p[0]) && p[1] == ':')
182:         p += 2;
183:     if (!((p[0] == '/' || p[0] == '¥¥') && p[1] == '¥0'))   検索のための
184:         strcat(p, "¥¥");                                     パスの作成
185:     p = strtail(path);
186:     strcpy(p, "*.*");
187:     if (srchfirst(path, &subdta)) {
188:         do {
189:             if (strequ(subdta.d_name, ".") ||
190:                 strequ(subdta.d_name, ".."))   "."と".."は無視
191:                 continue;
192:             strcpy(p, subdta.d_name);……………検索ファイル名の"*.*"の上に上書き
193:             srch_rec(&subdta); ……………………………自己再帰によって処理
194:         } while (srchnext(&subdta));
195:     }
196:     *p = '¥0';
197: }
198:
199: void    find(char *root)…………………スタートパスのタイプを判断して，ディレクトリの
200: {                                     検索を起動する関数
201:     DIR     dtabuf;
202:     int     pathtype;
203:     uchar   base[256];
204:     char    *filename();
205:
206:     if (exp_list == NULL)
207:         assert(0);                                          スタートパスとして
208:     if ((pathtype = filetype(root)) == PATH_NONE ||……許さないものの検出
209:         pathtype == PATH_ERROR || pathtype == PATH_AMBIGUOUS)
210:         fatal("Bad starting path <%s>.¥n", root);
```

```
211:        strcpy(path, root);
212:        jstrupr(path);
213:        if (pathtype == PATH_ROOTDIR)         ルートディレクトリの場合,
214:            set_root_dta(&dtabuf);            適当な情報を生成
215:        else
216:            if (!srchfirst(path, &dtabuf))            サブディレクトリ／ファイ
217:                fatal("Bad status <%s>.¥n", path);    ルの場合はsrchfirst関数に
218:                                                      よって情報を取得
219:        /* Avoid mysterious feature of MS-DOS function call */
220:        if (strequ(filename(path, base), ".") || strequ(base, ".."))
221:            strcpy(dtabuf.d_name, base);
222:        srch_rec(&dtabuf);......................処理ルーチンの呼び出し
223: }
224:
225: /* ---- end of exec.c ---- */
```

リスト4-20　EXEC.C

```
 1: /*
 2:  *  dirlib.c: library functions for directory access.
 3:  */
 4:
 5: #include <signal.h>
 6: #include <dos.h>
 7: #include <ctype.h>
 8: #include <string.h>
 9: #include <jstring.h>
10: #include "btype.h"
11: #include "misc.h"
12: #include "dir.h"
13:
14: #define ROOT_DATE    ((1 << 5) | 1)  /* 1/1/80 */...............ルートディレクトリの
15:                                                              タイムスタンプの定義
16: /*
17:  *  correct_e5(): correct bug of some MS-DOS version
18:  *  correct_e5(p);
19:  *  uchar    *p;     string to correct.
20:  *
21:  *  description:
22:  *    Some version of MS-DOS (correspond to Shift JIS) has bug
23:  *  in directory entry handler. When filename begin with '0xe5',
24:  *  ( Shift JIS code), it is converted to 0x05, but never recovered.
25:  *
26:  *  bugs:   This module support only Small model.
27:  */
28:
```

```
29: static  void    correct_e5(uchar *p)
30: {
31:     while (p != NULL) {
32:         if (*p == 0x05)
33:             *p = (uchar)0xe5;
34:         if((p = jstrmatch(p, "¥¥/")) != NULL)
35:             p++;
36:     }
37: }
38:
39: static  void    setdta(DIR *dta)
40: {
41:     union REGS regs;
42:
43:     regs.x.dx = (uint)dta;
44:     regs.h.ah = (uchar)0x1a;
45:     intdos(&regs, &regs);
46: }
47:
48: BOOL    srchfirst(uchar *path, DIR *dta)
49: {
50:     union REGS regs;
51:
52:     setdta(dta);
53:     regs.x.dx = (uint)path;
54:     regs.x.cx = 0x003f & ~DIR_VOL;
55:     regs.h.ah = (uchar)0x4e;
56:     intdos(&regs, &regs);
57:     if (regs.x.cflag == 0)
58:         correct_e5(dta->d_name);
59:     return regs.x.cflag == 0;
60: }
61:
62: BOOL    srchnext(DIR *dta)
63: {
64:     union REGS regs;
65:
66:     setdta(dta);
67:     regs.h.ah = (uchar)0x4f;
68:     intdos(&regs, &regs);
69:     if (regs.x.cflag == 0)
70:         correct_e5(dta->d_name);
71:     return regs.x.cflag == 0;
72: }
73:
74: void    set_root_dta(DIR *dta)
75: {
76:     dta->_attrib = DIR_DIR;
77:     dta->_ftime = 0;
78:     dta->_fdate = ROOT_DATE;        /* 1/1/80 */
79:     dta->_fsize = 0;
80:     strcpy(dta->d_name, "¥¥");
81: }
82:
```

……一部のMS-DOSのバグを避けるための処理

……findfirst関数とほぼ同じだが,
・検索属性は固定
・返値は成功のとき1，エラーのとき0

……ルートディレクトリの情報を適当に作成し，指定された領域に格納する関数．ここではタイムスタンプを1980年1月1日，0時0分0秒としている

```
 83: /*
 84:  *   getcdir(): get current directory of specified drive.
 85:  *   ret = getcdir(drive, buf);
 86:  *   uchar   drive;  1 for A, 2 for B ...
 87:  *   char    *buf;   buffer to save result
 88:  *   BOOL    ret;    FAIL for error, SUCCESS for success.
 89:  */
 90:
 91: static  BOOL    getcdir(uchar drive, uchar *buf)
 92: {
 93:     union REGS regs;
 94:
 95:     regs.h.dl = (uchar)drive;
 96:     regs.x.si = (uint)buf;
 97:     regs.h.ah = (uchar)0x47;     /* get current dir */
 98:     intdos(&regs, &regs);
 99:     if (regs.x.cflag)          /* Illegal drive number */
100:         return FAIL;
101:     correct_e5(buf);
102:     return SUCCESS;
103: }
104:
105: /*
106:  *   _chdir(): change current directory (keep current drive).
107:  *   ret = _chdir(path);
108:  *   int     ret;    -1 if error; 0 if success;
109:  *   char    *path;  path to set.
110:  */
111:
112: static  int _chdir(char *path)
113: {
114:     union REGS regs;
115:
116:     regs.x.dx = (uint)path;
117:     regs.h.ah = 0x3b;
118:     intdos(&regs, &regs);
119:     return regs.x.cflag ? -1 : 0;
120: }
121:
122: /*
123:  *   filetype(): check filetype.
124:  *   ret = filetype(path);
125:  *   int     ret;     (defined in misc.h)
126:  *   char    *path;  path to check.
127:  */
128:
129: int filetype(char *path)
130: {
131:     DIR     dtabuf;
132:     int     (*func)();
133:     uchar   drive;      /* 1 for A:, 2 for B:, .... */
134:     char    cwd[68];
135:     char    parent[5];
136:     char    *p;
```

91行目注：バッファbufに，driveで指定されたドライブのカレントディレクトリを格納する関数

112行目注：ドライブの変更なしでカレントディレクトリを変更する関数

129行目注：FILETYPE.C(リスト4-6)と同様のルーチン

```
137:     int      type;
138:
139:     if (jstrmatch(path, "*?[") != NULL) ············ワイルドカードは許されない
140:         return PATH_AMBIGUOUS;
141:     if (isalpha(path[0]) && path[1] == ':') {
142:         parent[0] = *path;
143:         parent[1] = ':';
144:         strcpy(parent + 2, "..");
145:         drive = toupper(*path) - 'A' + 1;
146:         cwd[0] = *path;
147:         cwd[1] = ':';                                    カレントドライブの保存
148:         cwd[2] = '¥¥';
149:         p = cwd + 3;
150:     } else {                    /* current drive */
151:         strcpy(parent, "..");
152:         drive = 0;
153:         cwd[0] = '¥¥';
154:         p = cwd + 1;
155:     }
156:     if (!getcdir(drive, p))
157:         return PATH_ERROR;·····················指定したドライブは存在しない
158:     func = signal(SIGINT, SIG_IGN); ····················CTRL+Cの割り込みの禁止
159:     if (_chdir(path) != 0) {/* cannot chdir; path isnot a dir */
160:         type = srchfirst(path, &dtabuf) ? PATH_REGULAR : PATH_NONE;
161:     } else {
162:         type = (_chdir(parent) == 0) ? PATH_SUBDIR : PATH_ROOTDIR;
163:         _chdir(cwd);            /* restore default dir */
164:     }
165:     signal(SIGINT, func); ····················割り込みベクタの復旧
166:     return type;
167: }
168:
169: /*
170:  *  filename(): strip directory part from path
171:  *  ret = filename(path, buf);
172:  *  char    *ret;       NULL for error, otherwise buf.
173:  *  char    *path;      Orginal path.
174:  *  char    *buf;       buffer to save result.
175:  */
176:
177: char    *filename(char *path, char *buf) ·····················pathで渡された文字列から,
178: {                                                            ドライブ名とディレクトリ部
179:     char    *p;                                              分を除いたものをbufにコピ
180:                                                              ーする関数
181:     if (path == NULL || buf == NULL) {
182:         if (buf != NULL)
183:             buf[0] = '¥0';
184:         return NULL;
185:     }
```

```
186:     if (isalpha(path[0]) && path[1] == ':')
187:         path += 2;
188:     for ( ;(p = jstrmatch(path, "¥¥/")) != NULL; path = p + 1)
189:         ;
190:     return strcpy(buf, path);
191: }
192:
193: /* ---- end of dirlib.c ---- */
```

リスト4-21　DIRLIB.C

```
 1: /*
 2:  *  docmd.c: command execution library for find.
 3:  */
 4:
 5: #include <stdio.h>
 6: #include <process.h>
 7: #include <stdlib.h>
 8: #include <string.h>
 9: #include "btype.h"
10: #include "misc.h"
11:
12: /*
13:  *  exec_cmd(): execute a command.
14:  *  ret = exec_cmd(argv, argc, pat);
15:  *  BOOL    ret;          return FALSE if error.
16:  *  char    **argv;       argument list.
17:  *  int     argc;         number of argument.
18:  *  char    *pat;         replaced string in exchange for "{}".
19:  *
20:  *  bugs:    exec_cmd() tries to execute external command (.COM
 and .EXE)
21:  *      befor shell internal command or batch command (.BAT), So it
22:  *      doesn't work well in the following case:
23:  *          1) external command has same name as internal command.
24:  *          2) if command line contain pipe or redirection.
25:  */
26:
27: BOOL    exec_cmd(char **argv, int argc, char *pat)
28: {
29:     char    **args;
30:     int     i, status, reason;
31:
32:     fflush(stderr);
33:     fflush(stdout);
34:     if ((args = xalloc(char *, argc + 2 + 1)) == NULL)
35:         fatal("Not enough core. Stop execution.");
36:     args[argc+2] = NULL;
37:     if ((args[0] = getenv("COMSPEC")) == NULL)
38:         args[0] = "command";
39:     args[1] = "/c";
```

子プロセスを起動するためのモジュール．EXECUTE.C(リスト4-9)とはコマンドラインの渡し方が異なる

コマンド行はcharへのポインタの配列として渡す．"{}"という引数は，pat(現在処理されているファイル名)で置換される

```
40:     for (i = 0; i < argc ; i++)
41:         args[i + 2] = strequ(argv[i], "{}") ? pat : argv[i];
42:     if ((status = spawnvp(P_WAIT, args[2], args + 2)) == -1)
43:         status = spawnvp(P_WAIT, args[0], args);
44:     free((char *)args);
45:     if ((reason = (status >> 8) & 0xff) != 0 && reason != 3)
46:         exit(1);      /* ^C quit, Critical device error, etc... */
47:     return (status & 0xff) == 0;
48: }
49:
50: /* ---- end of docmd.c ---- */
```

引数配列のコピーと，"{}"の置換

リスト 4-22 DOCMD.C

```
 1: /*
 2:  *   str.c:  string manipulate function.
 3:  */
 4:
 5: #include <ctype.h>
 6: #include <jctype.h>
 7: #include "btype.h"
 8: #include "misc.h"
 9:
10: /*
11:  *   match_name(): filename pattern match function for find.
12:  *   ret = match_name(fname, pat);
13:  *   BOOL    ret;          return TRUE, if match;
14:  *   char    *fname;       filename.
15:  *   char    *pat;         wild card pattern.
16:  *
17:  *    The following Meta-characters are support:
18:  *        [] * ?
19:  *    That like shell like interpretation.  Allow to use Kanji
    (Shift-JIS)
20:  *   in fname & pat.  Only 1-byte characters are recognzed Meta-
    character.
21:  *
22:  */
23:
24: static  kchar   getletter(uchar **p)
25: {
26:     uchar   *s;
27:     kchar   ch;
28:
29:     s = *p;
30:     if (*s == '¥0')
31:         return '¥0';
32:     if (iskanji(s[0]) && iskanji2(s[1])) {
33:         ch = s[0] << 8 | s[1];
34:         s += 2;
35:     } else
```

ワイルドカードのマッチングなどを行うモジュール

文字列pから1文字取得する関数

```
36:             ch = *s++;
37:         *p = s;
38:         return ch;
39: }
40:
41: static  BOOL    amatch(uchar *nam, uchar *pat)
42: {
43:         BOOL    match_name();
44:
45:         if (*nam == '¥0')
46:             return *pat == '¥0';
47:         do {
48:             if (match_name(nam, pat))
49:                 return TRUE;
50:         } while (getletter(&nam) != '¥0');
51:
52:         return FALSE;
53: }
54:
55: static  BOOL    gmatch(uchar *nam, uchar *pat)
56: {
57:         kchar   ch;
58:         kchar   pch;
59:         kchar   lch;
60:         BOOL    match;
61:         enum    { INIT, FIRST, HYPHEN } state;
62:         BOOL    match_name();
63:
64:         match = FALSE;
65:
66:         if ((ch = getletter(&nam)) == '¥0')
67:             return FALSE;
68:
69:         state = INIT;
70:         while (*pat != ']') {
71:             if ((pch = getletter(&pat)) == '¥0')
72:                 return FALSE;
73:
74:             switch (state) {
75:             case INIT:
76: first_char:
77:                 if (pch == '¥¥')
78:                     if ((pch = getletter(&pat)) == '¥0')
79:                         return FALSE;
80:                 match |= (ch == pch);
81:                 lch = pch;
82:                 state = FIRST;
83:                 break;
84:
85:             case FIRST:
86:                 if (pch != '-')
87:                     goto first_char;
88:                 state = HYPHEN;
89:                 break;
90:
```

41行目: ワイルドカード"*"を処理するルーチン

55行目: ワイルドカード"[……]"を処理するルーチン

```
 91:            case HYPHEN:
 92:                if (pch == '¥¥')
 93:                    if ((pch = getletter(&pat)) == '¥0')
 94:                        return FALSE;
 95:                match |= (lch <= ch) && (ch <= pch);
 96:                state = INIT;
 97:                break;
 98:            }
 99:        }
100:        if (!match)
101:            return FALSE;
102:        return match_name(nam, pat + 1); ………"]"の続きの部分は相互再帰によって処理
103: }
104:
105: BOOL    match_name(uchar *nam, uchar *pat)
106: {
107:        kchar   ch;
108:        char    *jstrchr();
109:
110:        while (TRUE)
111:            switch (ch = getletter(&pat)) {
112:            case '[':
113:                return gmatch(nam, pat);
114:
115:            case '*':
116:                while (*pat == '*')
117:                    getletter(&pat);
118:                if (*pat == '¥0')
119:                    return TRUE;
120:                return amatch(nam, pat);
121:
122:            case '?':
123:                if (getletter(&nam) == '¥0')
124:                    return FALSE;
125:                break;
126:
127:            case '¥0':
128:                return *nam == '¥0';
129:
130:            case '¥¥': …………………エスケープ文字
131:                if ((ch = getletter(&pat)) == '¥0')
132:                    return FALSE;
133:                /* fall through */
134:            default:
135:                if (getletter(&nam) != ch)
136:                    return FALSE;
137:                break;
138:            }
139: }
140:
141: /*
142:  *  jstrupr(): Kanji version of strupr().
143:  *
144:  *  ret = jstrupr(p);
145:  *  uchar   *ret;       same value p.
146:  *  uchar   *p;     string to convert.
147:  */
```

```
148:
149: uchar   *jstrupr(uchar *cp)
150: {
151:     register uchar  *p;
152:
153:     for (p = cp; *p != '¥0'; p++)
154:         if (iskanji(p[0]) && iskanji2(p[1]))
155:             p++;
156:         else
157:             *p = toupper(*p);
158:     return cp;
159: }
160:
161: /* ---- end of str.c ---- */
```

リスト4-23　STR.C

```
 1: /*
 2:  *  time.c: time handling module
 3:  *
 4:  *  BUGS: environment TZ is ignored.
 5:  */
 6:
 7: #include <dos.h>
 8: #include <time.h>
 9: #include "btype.h"
10: #include "misc.h"
11:
12: /*
13:  *  julius(): convert tm structured time info to time_t type.
14:  *  ret = julius(tp);
15:  *  time_t  ret;         time_t type data.
16:  *  struct  tm  *tp;     data to convert.
17:  */
18:
19: static  time_t  julius(register struct tm *p) ………tm構造体の内容をtime_t型に直すルーチン(gmtime関数の逆変換に相当)
20: {
21:     time_t  j;
22:     register int m, y;
23:     static int day_sum[] = { ………3月を基準にした，各月の1日の通年日
24:            0,  31,  61,  92, 122, 153,
25:          184, 214, 245, 275, 306, 337
26:     };
27:
28:     if ((m = p->tm_mon) < 0 || m > 11)
29:         return (-1L);           /* illegal */
30:     y = p->tm_year - 68;            /* since Mar. 1, 1968 */
31:     if (m >= 2)
32:         m -= 2;             /* Mar.-Dec. (2 - 11) to 0 - 9 */
33:     else {
34:         m += 10;            /* Jan.-Feb. (0 - 1) to 10 - 11 */
35:         y--;
```

```
36:     }
37:     j = y * 365 + y / 4;
38:     j += day_sum[m];
39:     j += p->tm_mday - 1;
40:     j -= 365 * 2 - 31 - 28;      /* adjust Jan. 1, 1970 */
41:     j *= 24;             /* unit in hour */
42:     j += p->tm_hour;
43:     j *= 60;             /* unit in minute */
44:     j += p->tm_min;
45:     j *= 60;             /* unit in second */
46:     j += p->tm_sec;
47:     return j;
48: }
49:
50: /*
51:  *  to_timet(): change format of timestamp from MS-DOS to Unix
52:  *      ret = to_timet(date, time);
53:  *      time_t  ret;
54:  *      unsigned short date, time;
55:  */
56:
57: time_t  to_timet(unsigned short date, unsigned short time)
58: {
59:     struct tm timebuf;
60:
61:     timebuf.tm_sec = (time & 0x1f) * 2;
62:     timebuf.tm_min = (time >> 5) & 0x3f;
63:     timebuf.tm_hour = (time >> 11) & 0x1f;
64:     timebuf.tm_mday = date & 0x1f;
65:     timebuf.tm_mon = ((date >> 5) & 0x0f) - 1;
66:     timebuf.tm_year = ((date >> 9) & 0x7f) + 80;
67:     return julius(&timebuf);
68: }
69:
70: time_t  gettime(time_t *p)
71: {
72:     union REGS regs;
73:     struct tm timebuf;
74:     time_t  result;
75:
76:     regs.h.ah = 0x2a;    /* get current date */
77:     intdos(&regs, &regs);
78:     timebuf.tm_year = regs.x.cx - 1900;
79:     timebuf.tm_mon = regs.h.dh - 1;
80:     timebuf.tm_mday = regs.h.dl;
81:
82:     regs.h.ah = 0x2c;    /* get cuurent time */
83:     intdos(&regs, &regs);
84:     timebuf.tm_hour = regs.h.ch;
85:     timebuf.tm_min = regs.h.cl;
86:     timebuf.tm_sec = regs.h.dh;
87:
88:     result = julius(&timebuf);
```

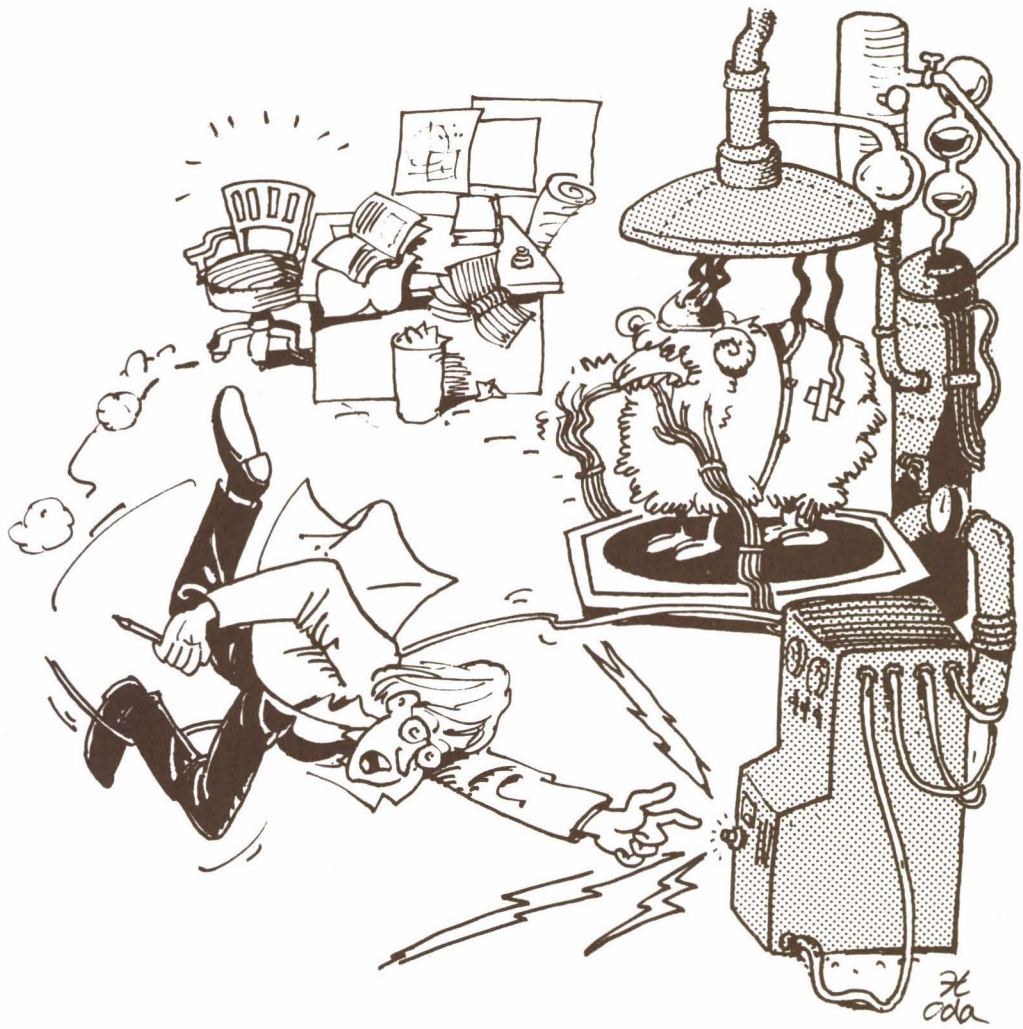
MS-DOSのタイムスタンプをtime_t型に変換するルーチン（57行目）

time関数とほぼ同等のルーチン．ただしTZは参照しない（70行目）

ファンクションコールによって現在の日付けと時刻を得て，julius関数によって変換する（76〜88行目）

```
89:         if (p != NULL)
90:              *p = result;
91:         return result;
92: }
93:
94: /* ---- end of time.c ---- */
```

リスト 4-24　TIME.C

第5章
割り込み処理

最近のパソコン上のＣコンパイラは，付属のライブラリ関数が充実し，高度なオプティマイズ機能により実行速度も向上しているため，Ｃ言語でもアセンブラなみの実行速度ときめの細かいプログラムの記述が可能になったということです．しかし，割り込みを使ったプログラムなどのようにハードウェアを直接扱うプログラムでは，やはりアセンブラの助けが必要になります．

この章では，ハードウェア割り込みの処理をＣ言語のプログラムでどのように記述すればよいかを解説します．そのサンプルとして，パソコン通信に使えるターミナルエミュレータを作ってみましょう．また，Ｃ言語プログラムとアセンブラプログラムのリンクについても見ていくことにします．

5.1 ハードウェアを直接扱うプログラム

■ アセンブラの代わりとしてのC言語

この章で紹介するプログラムは，従来アセンブラでなければ書けなかったプログラムをできるかぎりC言語で記述してみようという事例です．入門編，実習編でも解説してきたように，C言語それ自身は非常にシンプルな言語なので，ライブラリの助けがなければ文字1つを画面に出力することさえもできません．しかしその反面，ハードウェアをきめ細かく制御することが可能です．また，アセンブラのようにその記述のフォーマットに堅苦しい制限がないことから，アセンブラよりもわかりやすいプログラムを書くことが可能です．そのため，アセンブラの代わりの言語という使い方をした場合は，その記述性のよさ，可読性のよさから「ソフトウェア資源のむだをなくす」といった効果が期待できます．

■ C言語でアプリケーションプログラムを組む

最近のC言語の処理系はライブラリ関数などが充実し，また高度なオプティマイズをしてくれるため，多くのアプリケーションプログラムを組む場合，すべてをアセンブラで記述することは少なくなってきました．しかし，画面の高速な制御やハードウェアの割り込み処理などは，C言語だけで実用的なプログラムを組むことはむずかしいのが現状です．

とくに割り込みルーチンを書く場合は，全CPUのレジスタを待避しなければならないため，レジスタを直接操作できないC言語ではプログラムが書けません．また，ハードウェアの入出力を直接行う必要があるプログラムの一部では，「一定時間以内のタイミングで命令を送出しなければならない」というようなハードウェアの仕様に合わせたプログラム設計が必要なことがあります．C言語で用意されている関数で入出力を行うと，関数を呼ぶオーバーヘッドの時間がどうしてもかかるため，これらの要求には応えられなくなります．

このような場合は，アセンブラでハードウェアを直接扱うプログラムを記述し，C言語のプログラムとリンクするという手法がとられます．本来，ハードウェアの直接アクセスは，移植性の問題からやってはいけないことの1つですが，現状のパソコンで実用に耐えるプログラムを組む際にはどうしても必要になることが多く，避けては通れません．この章ではその実例として，パソコン通信に使えるターミナルエミュレータを取り上げます．このプログラムではアセンブラで記述する部分を最低限にとどめ，そのほとんどをC言語で記述することによって，多くの機種に簡単に移植ができるように配慮しています．

5.2 アセンブラとのリンク

C言語プログラムとアセンブラプログラムのリンクは，C言語そのものの知識だけでなく，以下の知識が必要になります．

・C言語プログラムはどのように動いているか
・アセンブラプログラムはどのように動いているか
・C言語プログラムとアセンブラプログラムのリンクとはどういうことか
・アセンブラプログラムとリンクするためのC言語プログラムの書き方はどうすればよいか
・C言語プログラムとリンクするためのアセンブラプログラムの書き方はどうすればよいか
・C言語プログラムとアセンブラプログラムのリンクの方法

これらのいずれも相互に関連があることなので，これからの解説はこのようにきれいには並んでいません．しかし以下の解説を読まれるまえに，これらのことがらをきちんと意識しておく必要があります．

■関数の呼び出し手順と変数の参照

C言語プログラムとアセンブラプログラムをリンクするという作業は，コンパイルの結果できたリロケータブル・オブジェクトファイルをリンカを使ってつなげるだけです．とはいっても，同一のプログラミング言語のように，単にオブジェクトファイルをリンクするだけでは，うんともすんともいいません．そこは「リンクを前提としたC言語プログラムの書き方」があり，「リンクを前提としたアセンブラプログラムの書き方」があるわけです．

C言語プログラムとアセンブラプログラムのリンクは，以下の事項に分類できます．

・C言語プログラムで定義した変数をアセンブラプログラムで参照／代入する
・アセンブラプログラムで定義した変数をC言語プログラムで参照／代入する
・C言語プログラムから，アセンブラプログラムのサブルーチン（C言語からみると関数）を呼ぶ（CALLする）
・アセンブラプログラムから，C言語プログラムの関数を呼ぶ（CALLする）

このほかにも考えればいろいろな場合がありますが，単にC言語プログラムからアセンブラプログラムにジャンプしたきりなどということはあまりないはずですから，上記の4つの場合を具体的に考えれば問題はないでしょう．

C 言語で作成した実行ファイルは，C コンパイラが翻訳したものなので，できあがった機械語プログラムはそのコンパイラ特有の〝規則〟があります．一方，アセンブラプログラムは，人間が書いたアセンブリ言語をそのまま(一対一で)機械語に置き換えているだけです．つまり，この両者をリンクする場合は，C 言語でコンパイルされた機械語の規則に合わせてアセンブラプログラムを書かなければなりません．

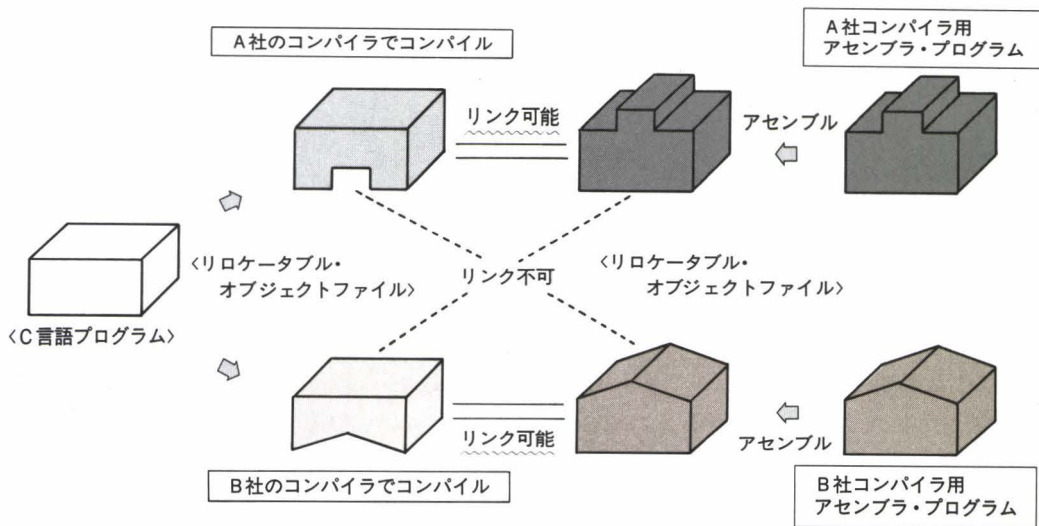



図 5-1　C 言語プログラムとアセンブラプログラムのリンク

このようなアセンブラプログラムを記述するには，C 言語で書かれたプログラムがどのような機械語にコンパイルされるのかという知識を得ておく必要がでてきます．しかも，やっかいなことに同じプログラムをコンパイルしても，C 言語の処理系によってその規則が異なるため，作成されたリロケータブル・オブジェクトファイルの形式は互換性がありません．そこで，以下では MS-C のラージモデルを例に，どのような機械語にコンパイルされるのかを見ていきましょう[*1]．

*1　基本的なスタイルは，どのメモリモデル，どの処理系でも同じだが，細部が異なっている．詳細は各コンパイラのマニュアルを参照のこと．

C 言語での関数

name_TEXT という名前のセグメント上のラベルつきの「サブルーチン」になる．つまり，関数の終わりには RETF 命令がある．name はそのコンパイル単位のファイル名になる(CLANG.C というファイル中の関数ならば，name は CLANG という名前になる)．

C 言語でのグローバル変数

DGROUP という名前のセグメント上のラベルつきの変数になる．

C 言語でのローカル変数

スタックセグメント上に配置される．

C 言語での関数名，変数名

C 言語の関数名や変数名のまえに「_」(アンダースコア)のついたものになる．つまり，func()という関数をアセンブラから参照する場合，そのラベル名は「_func」となる．

関数の引数

スタック上に逆順に(つまりソースファイル上の右側の引数から)PUSH される．このスタックポインタの調整は，呼ばれた側の関数が行うのではなく，呼んだ側が CALL 命令を終えたのちに行う．

返値(戻り値)

16 ビット長までのデータなら(ポインタ変数である／なしを問わず)AX レジスタに入れられる．32 ビットまでのデータであれば，AX レジスタに下位ワード，DX レジスタに上位ワードが入って返される．32 ビット以上のデータであれば，DX：AX レジスタにその値のあるアドレスが返ってくる．

resister 変数

最初の変数が SI レジスタにとられ，次の変数が DI レジスタにとられる．

最低限これらの規則を知っていれば，C 言語とリンクするアセンブラプログラムが書けるようになります．なお，C 言語のコンパイラは関数名や変数名の大文字と小文字を区別しますが，アセンブラではデフォルトの状態でこれらを区別しないようになっているのが一般的ですので注意してください．

■ C言語プログラムとアセンブラプログラムの記述の違い

以上の規則を実際にコーディングレベルで考えてみます．たとえば，**リスト 5-1** で示す C 言語で書かれた func() という関数は，**リスト 5-2** で示すアセンブラプログラムに相当します．前記の規則を見ながら，このプログラムを比較してみてください．

```
 1: int c = 0; ........................初期化されたグローバル変数C
 2:
 3: int fun(a,b)
 4: int a;  }
 5: int b;  } ............................引数はaとb
 6: {
 7: int d; ...............................ローカル変数d
 8:
 9: d = a + b + c;
10:
11: return(d); ........................dを返値とする
12: }
```

リスト 5-1　FUNC.C

```
 1: fname_TEXT  SEGMENT  BYTE PUBLIC 'CODE'  }
 2: fname_TEXT  ENDS                         } コードセグメント
 3:
 4: _DATA       SEGMENT  WORD PUBLIC 'DATA'  }
 5: _DATA       ENDS                         } データセグメント
 6:
 7: DGROUP      GROUP    _DATA                          セグメントのASSUME
 8:             ASSUME   CS: fname_TEXT, DS: DGROUP, SS: DGROUP,
    ES: DGROUP
 9:
10:
11: _DATA       SEGMENT        }
12:             PUBLIC   _c    } 変数Cは「_C」というラベルで，_DATAセグ
13: _c          DW       00H   } メント上に置かれる
14: _DATA       ENDS           }
15:
16:
17: fname_TEXT  SEGMENT        }
18:             PUBLIC   _func } 関数func()は「_func」というラベルで，
19: _func       PROC FAR       } FAR CALLで呼ばれるルーチン
20:
21:             push     bp    }
22:             mov      bp,sp } ベースポインタ，スタックポインタの退避
23:             sub      sp,2  }
24:
25:             push     di    }
26:             push     si    } レジスタ変数として使うSI，DIレジスタの退避
```

```
27:
28: ;    a = 6              value [a]'s offsets (argument 0)
29: ;    b = 8              value [b]'s offsets (argument 1)
30: ;    d = -2             value [c]'s offsets (local)
31:
32:             mov     ax,[bp+6]
33:             add     ax,[bp+8]
34:             add     ax,_c
35:             mov     [bp-2],ax
36: ;
37:             mov     ax,[bp-2]
38: ;
39:             pop     si
40:             pop     di
41:
42:             mov     sp,bp
43:             pop     bp
44:             ret
45:
46: _func       ENDP
47: fname_TEXT  ENDS
48: END
```

計算部分

返値のセット（AXレジスタに置かれる）

退避したレジスタをもとに戻す

d	←BP−2
CallでPUSHされたアドレス	←BP
bp	←BP÷4
a	←BP÷6
b	←BP÷8

リスト5-2　FUNC.ASM

■インライン・アセンブル機能を持ったC言語の処理系

C言語プログラムとリンクするアセンブラプログラムを作った場合，それぞれを別々に管理しなければならないという問題が起こります．こういったやっかいな問題を避けるために，ほとんどのC言語の処理系にではインライン・アセンブル機能が用意されています．これは，C言語プログラムのソースリスト中にそのままアセンブラ命令を書いておき，コンパイルできる機能のことです．

ただし，プログラムによってはこれもよしあしでしょう．たとえば，ふだん使い慣れているアセンブラの書式とインライン・アセンブラの書式が違うといったことに悩まされる場合や，インライン・アセンブラ自身がコンパイラのなかに組み込まれていると，そのバグなどに出会ったら万事休すということもあります．逆に，ちょっとしたCプログラムにアセンブラプログラムを加える際にはまことに重宝します．

また，C言語プログラムのコンパイル結果をアセンブラのソースリストとして出す機能があります．この場合ファイル管理に少々難があるものの，使い慣れたアセンブラを使ってアセンブルできるという安心感があります．

5.3 割り込み処理の概念

割り込み処理は，もともと決められた１つのタスクを順番に処理していくことしかできないストアド・プログラム方式(プログラム記憶方式)のハードウェアの欠点を補うために生まれたもので，ソフトウェアとハードウェアが密接にからんだアーキテクチャです．この節では，ハードウェア割り込みの概念について解説しましょう．

■ 割り込み処理とは?

パソコンにおける割り込み処理は，大きく分けて次の２つに分類されます．

- **ハードウェア割り込み**
- **ソフトウェア割り込み**

後者の割り込みは〝割り込み〟というより，むしろ〝サブルーチンコール〟に近いものです．つまり，プログラムの任意の場所でINT命令を発行すると，その割り込み番号にしたがったプログラムに飛んでいき，そのプログラム中のリターン命令で，さきに実行したINT命令の次の命令から再び実行するというものです．これは第4章でも取り上げたように，MS-DOSのシステムコールに使われています．INT命令でコールするとCALL命令と違って，CALLする側がFARであろうとNEARであろうと関係なくジャンプできます．

したがって前者の割り込みが，実は本来の意味での割り込み処理です．つまり，読んで字のごとく，通常の処理の最中に別の処理が「割り込む」という操作を可能にしているCPUの機構そのものを意味します．

ハードウェア割り込みのメカニズムは，次ページの**図5-2**のようになっています．この図を見ればわかるように，ハードウェアから直接CPUに割り込みがかかることを除けば，その処理の流れ自体はソフトウェア割り込みとまったく同様です．ただし，OSのシステムコールのように，すでにあるものを利用するのではなく，自前で各種の設定をしなければなりません．

割り込み処理を行う場合は，その処理を行うプログラム(サブルーチン)がどこにあるのかをあらかじめCPUに知らせておきます．割り込み処理は，現在進行している最中のプログラムを途中で止めて行う処理なので，迅速に，そして割り込まれたプログラムにはなんの影響もないように作成しなければなりません．また，割り込み処理の最中に，別のあるいは同一の割り込み処理が重ねて発生した場合はどうするのかということも考えに入れておく必要があります．

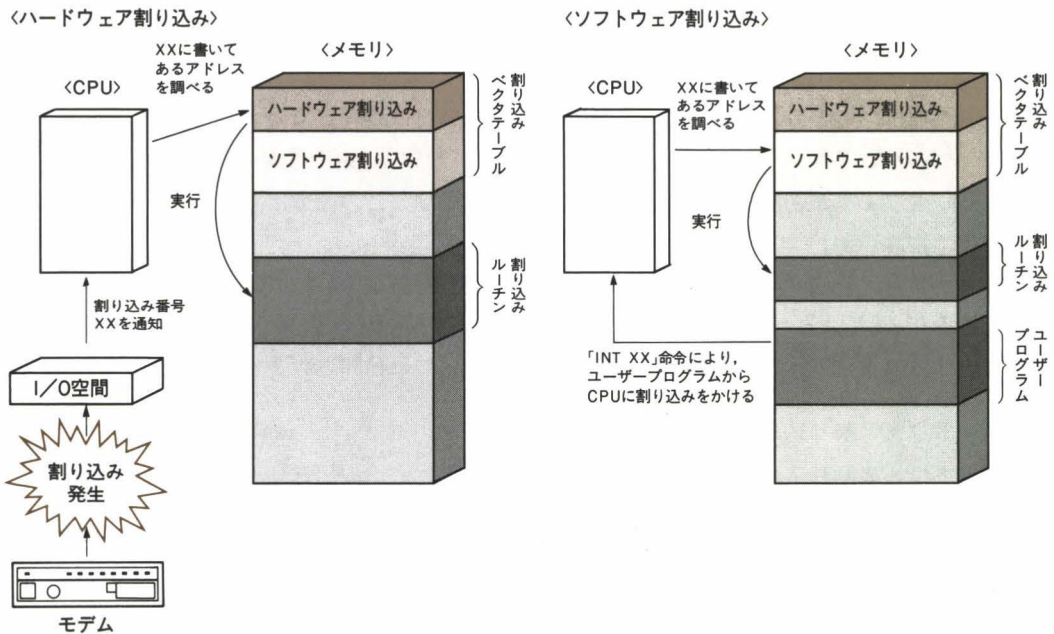



図5-2　ハードウェア割り込みとソフトウェア割り込み

■ハードウェア割り込みを利用したターミナルエミュレータ

ハードウェア割り込みの実例として，ここではRS-232Cの割り込みを使ったターミナルエミュレータを作成してみます．パソコンをターミナルとして使う場合，必要なターミナルエミュレータの条件は次のとおりです．

・必要な通信速度で，受信される文字データの落ちがない
・必要な通信速度と打鍵速度で，送信される文字データの落ちがない

こういった条件を満たすプログラムを作る場合，いちばん気をつけて作成する必要があるのはなんといっても文字データの受信です．文字データはいつ向こうからやってくるかわかりません．これが送信データであれば，ターミナルエミュレータ側がイニシァチブをとってその送信のタイミングを決定できますが，受信データの場合はそうとはかぎりません．たとえばディスクアクセスの最中にやってくるデータがあるかもしれないのです．

そこで，ターミナルエミュレータを作る場合，送信側はともかく，受信側は「割り込み」という手法を使うのが普通です．つまり，たとえパソコンがどんな仕事の最中でも，文字データがやってくると，一時的にその仕事をやめて文字データの受信を行い，それが終わるとまたもとの仕事に戻るという割り込み処理ルーチンを作り，文字データの落ちがないようにします．

このように割り込みが起こったときに起動され，処理を行うプログラムを**割り込みサービスルーチン**と呼びます(図 5-3)．

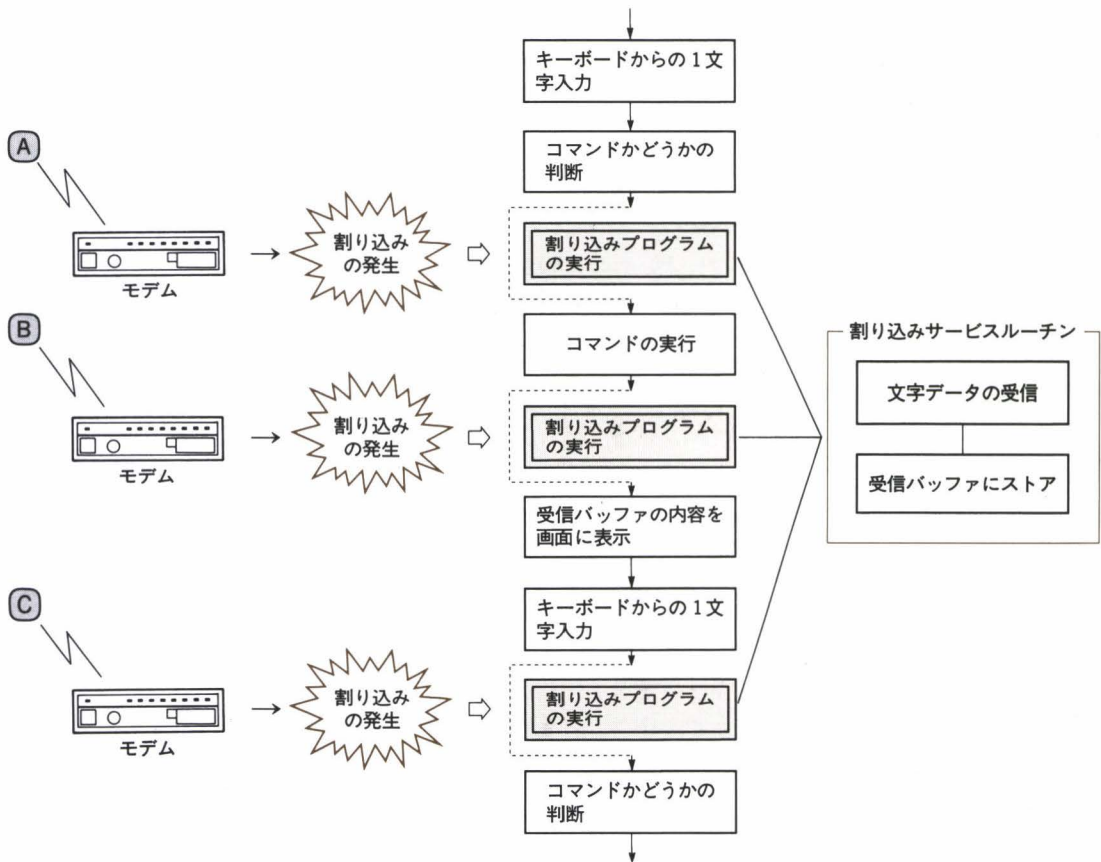



図 5-3　通信での割り込み処理

割り込み処理はその性質上，一度セットアップされると，現在動いているプログラムとは無関係に実行されます．したがって〝裏でプログラムが勝手に動いている〟といったかたちになります．割り

込まれる方の〝現在動いているプログラム〟は，今回のサンプルプログラムでいえば，「ターミナルエミュレータ本体」かもしれないし，「MS-DOSのキーボード入力処理ルーチン」かもしれません．また，ひょっとすると，「割り込みサービスルーチンの最中」かもしれません．つまり，自分で作成したプログラムだけが走っている環境で，これから作ろうとしている「割り込みサービスルーチン」は動くわけではないのです．したがって，細心の注意を払って〝他人の作ったプログラムに迷惑をかけないように〟割り込みサービスルーチンを書き，正しくセットアップする必要があります．

■ C言語での割り込み処理の扱い

通常の割り込みサービスルーチンは，たいていアセンブラのみで書かれます．これは，前述したように〝割り込み〟という性質から，次のような要求があるためです．

- 処理はなるべく迅速でなければならない
- 割り込まれたプログラムに影響を与えてはならない

主に前者の理由によって，割り込みサービスルーチンはアセンブラで書かれることが多いわけです．しかし逆にいえば，速度やほかのルーチンへの影響がなければ，割り込みサービスルーチンをC言語で書くことも可能です．ここでは「C言語で割り込みプログラムを記述する」という主旨から，普通はあまりやらない〝そのほとんどが〟C言語で書かれた割り込みプログラムを作成してみます．もちろん，できあがったプログラムは十分実用に耐えるものです（実際に19200bpsの送受信が可能*2）．

しかし，前節でも触れたように，割り込みサービスルーチンの〝すべて〟をC言語で書くことはできません．それは割り込まれるまえの環境を保存しておくために，すべてのレジスタを待避させなければならないからです（C言語では，すべてのレジスタを直接扱うことはできない）．そこで，割り込みがかかったときに呼ばれるアセンブラルーチンでは，すべてのレジスタを待避させてから，C言語のルーチンを呼べるようにレジスタなどをセットアップし，実際に割り込み処理を行うC言語の関数を呼び出します．そして，C言語の関数から返ってきたら，レジスタをもとの状態に戻してIRET命令を発行し，割り込まれた直後の状態に戻せばよいわけです．こうすることによって，最小限のアセンブラを記述するだけで割り込みサービスルーチンを作成できます．次ページの**図5-4**に割り込みサービスルーチンの構造を示しておきます．

この手法は割り込みに限ったことではなく，異なる言語間でのインターフェイスをとる場合にもよく使われる手法です．たとえばFORTRANとPascalでは機械語に落ちたときの呼び出し手順が違いますが，これらのあいだを取り持つのにアセンブラルーチンを入れ，Pascalのプログラムからはあたかも Pascalそのもののルーチンを呼んでいるように見せ，またFORTRANからは相手のPascalのルーチンがあたかもFORTRANそのものに見えるようにするといったやり方です．

---

*2　CPUのクロックが8MHzの場合，クロックの分周比（204ページ参照）の関係上，9600bpsまでとなる．

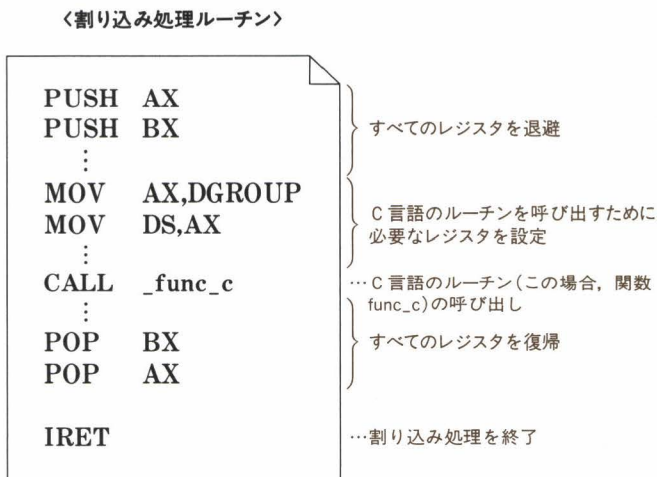



図 5-4 割り込みサービスルーチンの構造

■ 割り込み処理の記述

割り込みプログラムの動作の概略は理解しても，割り込み処理を実際に記述するためには，以下のような情報が必要です．

割り込みサービスルーチンのセットアップ

割り込み処理では，あらかじめセットアップされた割り込みサービスルーチンを組み込んでおかなければならないが，それはどのようにセットされるのか？

割り込みサービスルーチンの呼び出し方法

割り込みサービスルーチンに外部の割り込みがはいった場合，どのようにそのルーチンが呼ばれるか？

割り込みサービスルーチンの記述

割り込みサービスルーチンはどのような規則や制限のもとで書かなければならないか？　とくにハードウェアに密着した部分はどう記述するのか？

「割り込み」はもともとなにか別の仕事の最中に起こるものですから，このルーチンをほんの少しでも間違えると，コンピュータは割り込みの仕事から返ってきたときに割り込まれるまえにやっていた仕事の全部，あるいは一部を忘れてしまうことも起こりえます．こういった場合のほとんどが，〝暴

走″という事態を引き起こします．つまり，割り込みを使ったプログラムはデバッグをしようにもデバッグできないという事態に陥ることがかなりあるのです．これが「割り込みを使ったプログラムはむずかしい」といわれる理由です．

■ 必要なハードウェアの知識

ハードウェア割り込みを考えるうえでとにかく必要なのは，ハードウェアそのものの知識であることはいうまでもありません．ここではハードウェアがどのようにCPUに接続され，CPUはどのようにハードウェアをコントロールするのかを考えてみます．

80系CPUではCPUにつながっている周辺機器を制御するために，メモリエリアとは別にI/Oアドレス空間を持っています(**I/Oマップド I/O方式**)．これに対して68系CPUは，メモリエリアに周辺機器を制御する特殊なアドレス空間を設けています(**メモリマップド I/O方式**)．両方式の利点と欠点を図5-5に示しておきます．

〈I／Oマップド I／O方式〉

アドレスバス
各種周辺機器
CPU
メモリエリア
I/O エリア
データバス

メモリエリアとI／Oエリアが分かれているために，I／Oエリアをアクセスするための別の命令が必要になる．しかしI／Oエリアが別になっているためハードウェアが接続しやすい

〈メモリマップド I／O方式〉

アドレスバス
各種周辺機器
CPU
メモリエリア
I／Oエリア
メモリエリア
データバス

I／Oエリアが通常のメモリエリアの一部に割り当てられているため，メモリを扱うのと同じ方法でアクセスできる．逆に言えば，誤ってI／Oエリアをアクセスしてしまう可能性がある

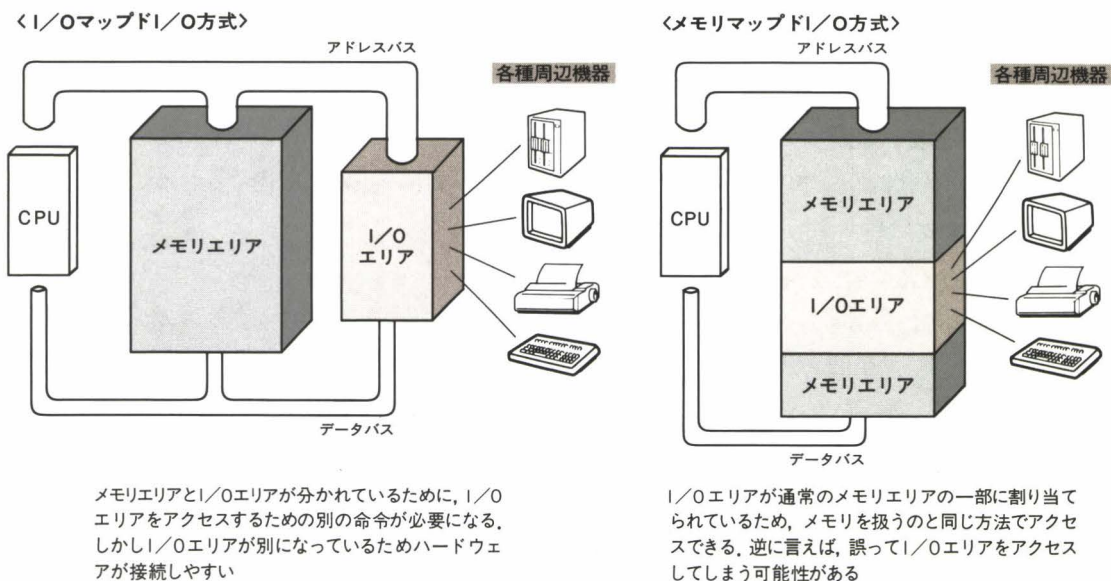



図5-5　I/Oマップド I/O方式とメモリマップド I/O方式

I/Oマップド I/O方式では，「I/Oアドレスのxx番地にはxxというハードウェアのxxレジスタがある」というように，その場所と役割が決まっています．たとえば，今回操作するRS-232Cコントローラi8251のようなUSART(Universal Synchronous Asynchronous Receiver Transmitter，直訳すると「多目的同期非同期送受信機」となる)は，「xx番地にはRS-232Cコントローラのxxレジスタがあり，その第nビットを0にすると，このような動作をし，1にするとこのような動作をする」といった特定の働きをします．このほかにも，i8259割り込みコントローラ(PIC：Programmable Interrupt Controller)，i8253プログラマブル・カウンタ，i8255パラレルインターフェイス，さらにはまったくオリジナルなハードウェアまで実にさまざまな周辺機器が接続されています．

ここでは，PC-9801シリーズにおける「割り込み」，とくにRS-232Cに関する処理を行うのに必要な個々のハードウェアの関連図を次ページの**図5-6**に示しておきます．

プログラムを作る側では，これらのハードウェアを熟知したうえでプログラムを組む必要がありますが，ここでは残念ながらページ数の都合でその一部しか解説できません．ハードウェア割り込みに関するプログラムをいちから組む場合は，ハードウェアの知識がかならず必要になりますから，しっかりと勉強しておきましょう．通常は，メーカーから発売されているハードウェアの資料を取りよせて調べることから始まります．しかし多くの場合，理屈抜きに，「このハードウェアはこのような仕様だからこう書く必要がある」と書いてある場合がほとんどです．

ハードウェアの仕様を理解できれば，その仕様にしたがってデータの読み込みや書き込みなどの動作を記述していくことになります．こういったハードウェアの制御のためのI/Oアドレスへの書き込みはアセンブラの「OUT命令」で行い，読み込みは「IN命令」で行います．このOUT命令とIN命令をC言語から直接扱えるようにしたライブラリ関数がoutp関数とinp関数です．C言語からのハードウェアのコントロールはこの2つの関数を使うことで実現できます．

関数名	書　式	返　値	機　能
inp( )	int inp(port); 　unsigned int port;	ポートから入力した値	入力ポートportから入力する
outp( )	int outp(port, value); 　unsigned int port; 　int value;	valueの値	出力ポートportへvalueの値を出力する

表5-1　outp関数とinp関数の仕様(MS-C)

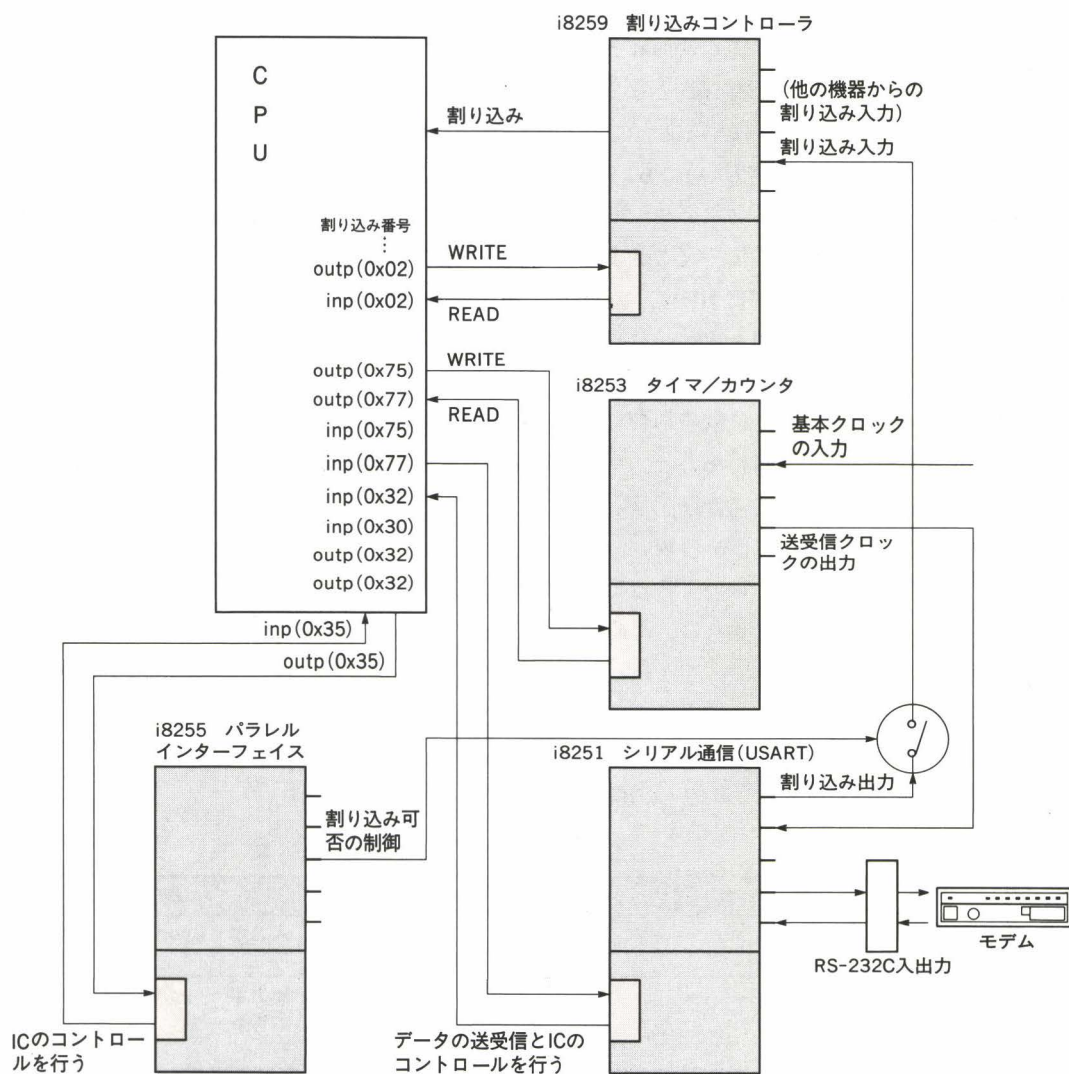



図5-6　PC-9801シリーズのハードウェアの接続図(RS-232C関連)

5.4 割り込みプログラムの実際

パソコンなどのコンピュータの割り込みのメカニズムはそのCPUごと，ハードウェアごとに違います．したがって，ここでの説明もある特定のハードウェアに限定して行わなければなりません．ここでは，8086系CPUを使った日本電気のPC-9801シリーズを例に，より具体的な割り込みプログラムの記述について解説しましょう．ほかのパソコンのハードウェアであっても，やっていることは同じですから，ここでの説明をよく理解したうえで，同じような手順を踏んでインプリメントしてみてください．

■ 8086系CPUのハードウェア割り込みのメカニズム

8086系CPUでは，割り込みが発生すると，発生した割り込みの種類1つ1つに割り当てられた「割り込み番号」がCPUに伝えられます．この番号は「この割り込みならxx番」というように決まっていますが，ハードウェアによっては変更可能なものもあり，プログラム自身からこの割り込み番号が変えられる場合もあります．

次ページの**表5-2**は，PC-9801シリーズの割り込みベクタテーブルの表です．**インタラプトベクタ**の割り当ては，システムセットアップルーチンによってソフトウェア的に決められるものですから，必要に応じてベクタテーブルを書き換えることができます．今回のプログラムでも，その方法を使ってシステムで使用しているはずのベクタテーブルをプログラム用に書き換えて使っています．また，このベクタテーブルは，必要に応じてその順序を変更すること(インタラプトのローテーション)も可能です．

199ページの**図5-7**は，8086系CPUにおける割り込み機構を図示したものです．このプログラムで使っているRS-232Cであれば割り込み番号は「0x0C」ですから，割り込みがかかるとまず割り込みコントローラに割り込み番号が伝えられます．そして割り込みコントローラからCPUに割り込み番号が伝達され，割り込み番号(0C)×4番地(＝絶対番地)，すなわちセグメント0000，オフセット0030(これを通常0000：0030と書く)のアドレスから書かれている4バイトの値のアドレス(インタラプトベクタという)のところにジャンプし，制御をそのプログラムに移します．この4バイトのアドレス値は低いアドレスから，オフセット下位1バイト，オフセット上位1バイト，セグメント下位1バイト，セグメント上位1バイトの順に書かれています．この際，CPUは「FAR CALL」のように，割り込みが起こったときに実行していたアドレスをスタック上に自動的にPUSHします．つまり，割り込み処理が終わったときに返るべきアドレスをセーブしておくわけです．このセーブされた番地は割り込みサービスルーチンのIRET命令によって，再び「CS：IP」にロードされ，プログラムの実行がそこから再開されます．

ベクタアドレス	ベクタ番号	用　　途	備　　考
0-3	0	除算エラー	CPU自身による割り込み
4-7	1	シングルステップ	
8-B	2	NMI	
C-F	3	INT 3	
10-13	4	オーバーフロー	
14-17	5	ハードコピー(COPY)キー	ソフトウェア割り込み
18-1B	6	STOPキー	
1C-1F	7	インターバルタイマ	
20-23	8	タイマ	ハードウェア割り込み
24-27	9	キーボード	
28-2B	A	CRTV(V-SYNC)	
2C-2F	B	拡張バスINT 0	
30-33	C	RS-232C	………今回使う割り込み
34-37	D	拡張バスINT 1(CMT)	
38-3B	E	拡張バスINT 2(ODAプリンタ)	
3C-3F	F	システム予約	
40-43	10	セントロプリンタ	
44-47	11	拡張バスINT 3(HD)	
48-4B	12	拡張バスINT 41(640KバイトFD)	
4C-4F	13	拡張バスINT 42(1MバイトFD)	
50-53	14	拡張バスINT 5	
54-57	15	拡張バスINT 6	
58-5B	16	8087	
5C-5F	17	ノイズ(システム予約)	ソフトウェア割り込み
60-63	18	KB, CODE-CRT, GRAPH-CRT	
64-67	19	RS-232C	
68-6B	1A	カセット, プリンタ	
6C-6F	1B	DISK/BIOS(1MバイトFD, 640KバイトFD, HD)	
70-73	1C	カレンダ, インターバルタイマ	
74-77	1D	システム予約	
78-7B	1E	N_{88}-BASIC	
7C-7F	1F	システム予約	
80～FF	20～3F	システム予約	
100～1FF	40～7F	ユーザー用	
200～3FF	80～FF	システム予約	

(「PC-9801シリーズ テクニカルデータブック」アスキー出版局発行をもとに作成)

表 5-2　PC-9801 シリーズの割り込みベクタ

このような割り込み機構には，CPU だけではなく，前述した i8259 のような割り込みコントローラと呼ばれる IC も必要です．割り込み処理を行うためには，この割り込みコントローラが不可欠なので，最近の CPU ではこの IC が最初から CPU のチップに含まれていることが多くなりました．

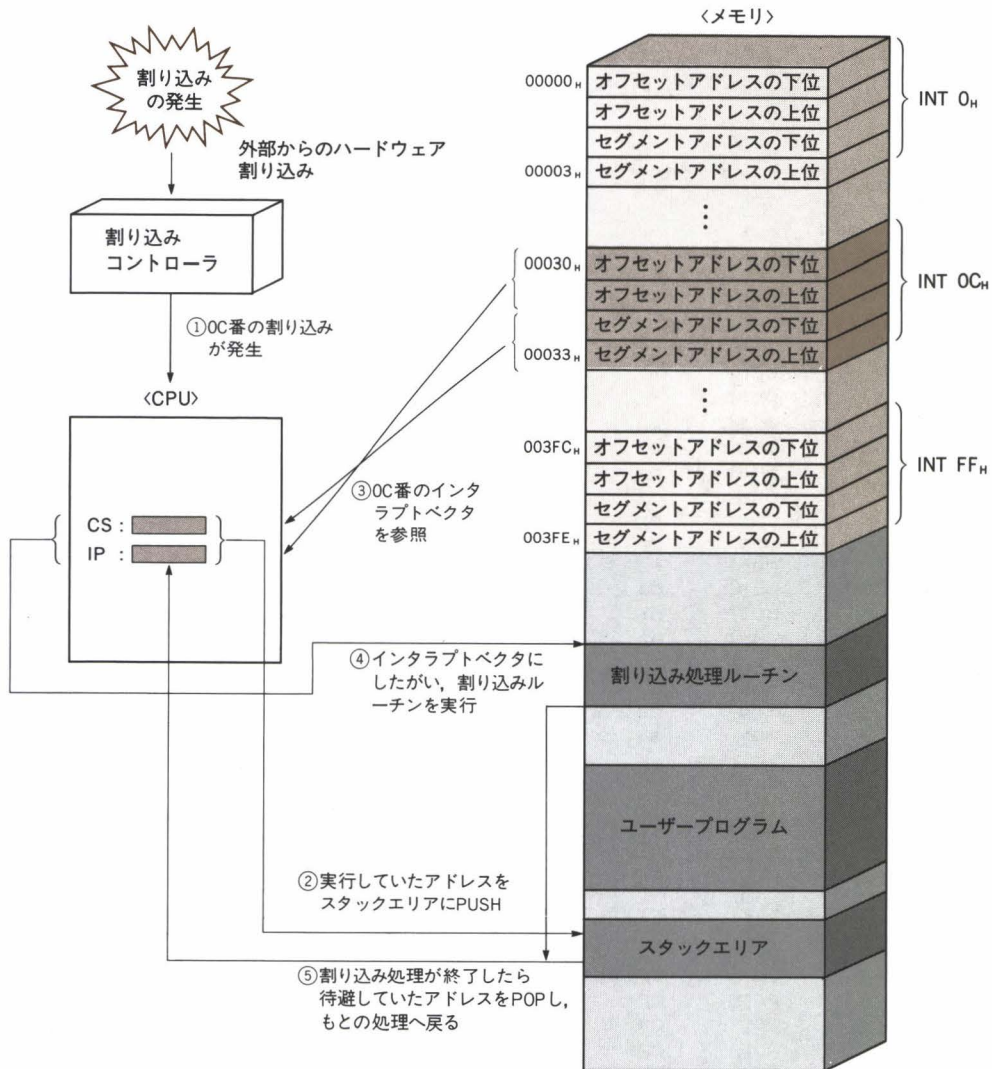



図 5-7　8086 系 CPU の割り込み動作

■ 割り込みサービスルーチンの記述

割り込み処理で起動される割り込みサービスルーチンは，今回のプログラムでは RSINT.ASM のなかの_rsintr というラベルから始まる一連のアセンブラルーチンです．このプログラムは，すべての

レジスタを待避し，C言語の関数を呼ぶためのセットアップを行って，rsintc関数を「FAR CALL」します．割り込み処理から戻ってくると，レジスタを復帰し，割り込みまえの処理に戻ります．

ここでわかるように，実際の割り込み処理はCALL命令で呼び出されるC言語ルーチン(rsintc関数)で行われており，このアセンブラルーチンではなにもしていません．

```
27: ;
28: ;    Interrupt CALLed with RS-232C
29: ;
30:             PUBLIC  _rsintr
31: _rsintr     PROC FAR
32: ;
33:             cli
34:             pushf
35:             push    ax
36:             push    bx
37:             push    cx
38:             push    dx
39:             push    bp
40:             mov     bp,sp
41:             push    si
42:             push    di
43:             push    ds
44:             push    es
45: ;
46:             mov     ax,DGROUP
47:             mov     ds,ax           ; Set DS: to DATA-SEGMENTS
48:             mov     es,ax
49: ;
50:             cld                     ; Clear Direction Flag
51: ;
52:             call    _rsintc         ; Call C-language Routine
53: ;
54:             pop     es
55:             pop     ds
56:             pop     di
57:             pop     si
58:             mov     sp,bp
59:             pop     bp
60:             pop     dx
61:             pop     cx
62:             pop     bx
63:             mov     al,20h
64:             out     0,al
65:             pop     ax
66:             popf
67:             sti
68:             iret
69: ;
70: _rsintr     ENDP
```

リスト5-3　RSINT.ASM ①

このプログラムの動作を実際に追ってみましょう．まず，33行目でCLI命令を発行し，ほかからの割り込みを止めています．34行目から44行目ですべてのレジスタをPUSHしてスタックエリアに退避します．

46行目から50行目の処理はC言語ルーチンを呼ぶためのセットアップです．MS-Cではディレクションフラグは常に0でなければならないのでフラグをクリアします．C言語ルーチンのメモリモデル*3(ラージモデル)では，DSには「DGROUP」という値がはいっていなければならないので，次にそのための処理をします．

このように，C言語ルーチンを呼び出すための準備が整ったら，52行目でコールします(C言語の割り込み処理関数については次節で解説する)．割り込み処理が終わったら，さきにPUSHしたレジスタの内容を復帰します(54行目から66行目)．63，64行目は，割り込みコントローラi8259にEOI(End Of Interrupt)命令を発行し，68行目のIRET命令で割り込みまえの処理に戻ります．

```
71: ;
72: ;          INTERRUPT CLEAR FUNCTIONS
73: ;
74: PUBLIC          _cli
75: _cli            PROC FAR
76: ;
77:                 cli
78:                 ret
79: ;
80: _cli            ENDP
81: ;
82: ;          INTERRUPT SET FUNCTIONS
83: ;
84: PUBLIC          _sti
85: _sti            PROC FAR
86: ;
87:                 sti
88:                 ret
89: ;
90: _sti            ENDP
```

リスト5-4　RSINT.ASM ②

74行目から始まっている「_cli」というルーチンと，84行目から始まっている「_sti」というルーチンは，C言語のライブラリにはないアセンブラのCLI(CLear Interrupt enable flag：外部割り込みの禁止)命令とSTI(SeT Interrupt enable flag：外部割り込みの許可)命令を実行するためのルーチンで，C言語ルーチンから呼ばれるために作ってあります．この2つの命令はフラグ以外のレジスタをいっさい壊さないので，レジスタの退避は行っていません．

＊3　メモリモデルについては第1章を参照．

■ 割り込みサービスルーチンのセットアップ

前述したアセンブラルーチンはRS-232Cから割り込みがかかると起動され，その処理をC言語で書かれたルーチンに渡し，そこで割り込み中に必要な処理を行い，またこのアセンブラルーチンに戻ったあと，割り込み処理を終了します．これらの呼び出し関係を**図5-8**にまとめてみましょう．

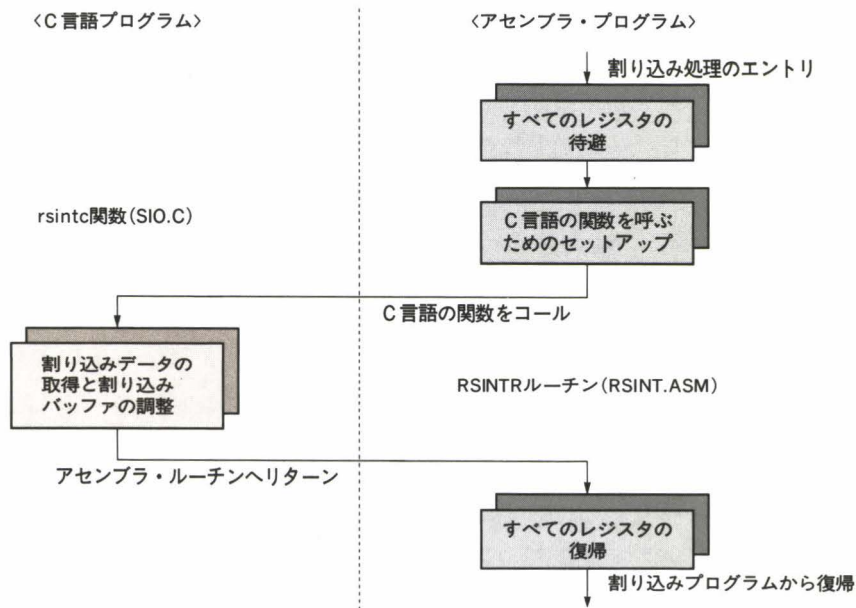



図5-8　割り込みプログラムの呼び出し関係

さて実際には，アセンブラの割り込みルーチンが呼ばれるまえにいくつかセットアップをしておかなければなりません．このセットアップには大きく分けて以下の3つの部分があります．

- 割り込みを発生するハードウェアの設定
- 割り込みが起こったときに割り込みルーチン(_rsintr ラベル：RSINT.ASM)に制御を移すための設定
- 割り込みが起こったとき処理するルーチン(rsintc 関数：SIO.C)の内部で使われるパラメータの初期化

これらの設定や初期化を行っている関数はC言語で書かれているsio_init関数(SIO.C)です．このルーチンを実際に追ってみましょう．

```
 96: /* ****************************************************************
 97:      time wait functions for OUTP() & INP()
 98: **************************************************************** */
 99:
100: void    io_wait()
101: {
102: int     i;
103: ;
104: }
105:
106:
107: /* ****************************************************************
108:      Get SIO-Buffer Dynamically
109: **************************************************************** */
110:
111: BOOL    set_bufsiz()
112: {
113: size = 0xFFF0;
114: while((char *)NULL == (siobuf = (char *)malloc(size)))
115:      {
116:      size /= 2;
117:      if(0x100 > size) return(FALSE);
118:                                 /* Not enough memory Space */
119:      }
120:
121: if((char *)NULL == (siobuf = (char *)realloc(siobuf,size / 2)))
122:      return(FALSE);
123:
124: txbsiz = 0xFFF0;
125: while((char *)NULL == (txbuf = (char *)malloc(txbsiz)))
126:      {
127:      txbsiz /= 2;
128:      if(0x100 > txbsiz) return(FALSE);
129:                                 /* Not enough memory Space */
130:      }
131:
132: if((char *)NULL == (txbuf = (char *)realloc(txbuf,txbsiz / 2)))
133:      return(FALSE);
134:
135: size   /= 2;
136: txbsiz /= 2;
137:
138: return(TRUE);
139: }
140:
141:
142: /* ****************************************************************
143:      Initial SIO
144: **************************************************************** */
145:
146: int     sio_init(int speed, int length, int parities, int stops,
    int xc)
147:      /* speed    : Baud-rates      /[0-9] for [75bps-38400bps] */
148:      /* length   : Data-bit length/[0-3] for [5bits-8bits] */
149:      /* parities : Parity Bits     /[0,2=NO / 1=Odd / 3=Even] */
```

```
150:     /* stops     : Stop Bits       /[1=1bit / 2=1.5bits / 3=2bits] */
151:     /* xc        : x-flow control /[0=OFF / 1=ON] */
152: {
153: int      i, j;
154:
155: if(!set_bufsiz()) return(FALSE);
156: cli();
157: p = (char *)0x00000501L;
158: if((0x10 == (0x70 & *p)) || (0 == (0x80 & *p)))
159:     i = 0;
160: else
161:     i = 1;
162:
163: io_wait();
164: outp(SYS_CNT_MOD,0xB6);
165: io_wait();
166: outp(SYS_CNT_SET,speed_tbl[i][speed]);
167: io_wait();
168: outp(SYS_CNT_SET,speed_tbl[i][speed] >> 8);
```

リスト5-5　SIO.C ①

sio_init 関数は SIO.C の 146 行目から始まります．引数は 147 行目から 151 行目に説明があります．これらのパラメータのうち speed と xc を除いたものは，すべて通信ハードウェアの本体である i8251 にセットされます．

155 行目では，まず set_bufsiz 関数を使って送受信の割り込みバッファをメモリ上にダイナミックに割りつけています．この関数は 111 行目から始まり，malloc 関数を使ってメモリ領域を確保します．送信，受信ともおよそ 32K バイトのエリア(最大)を取れるようになっており，使用可能なメモリによってその大きさが変化します．一度割りつけたエリアをもう一度半分の大きさで realloc しているのは，最初に取ったバッファエリアでメモリが一杯になってしまい，次のメモリエリアが取れないためにプログラムが起動しないという事態を避けるためです．

通信速度(bps)	クロック5/10MHzの分周値	クロック8MHzの分周値
38400	使用不可	使用不可
19200	8	使用不可
9600	16	13
4800	32	26
2400	64	52
1200	128	104
600	256	208
300	512	416
150	1024	832
75	2048	1664

表5-3　クロック周波数による分周値

156 行目では cli 関数で CLI 命令を発行し，ハードウェア割り込みがかからないようにしたあと，割り込み関係のセットアップを行う準備をしています．

157 行目から 161 行目では，PC-9801 の機種の設定を見て，CPU のクロックを 5／10MHz か，8MHz かを判断しています．PC-9801 では CPU のクロックから分周して RS-232C の通信速度を決めるクロックを得ています．もちろん，ほかのパソコンでは必要ないものもあります．ここで決められた i の値が，通信速度を割り出すときに，クロック周波数の分周値のテーブルの選択をします．

163 行目から 168 行目は，i8253 カウンタ・ハードウェアを 1/n 分周器として通信速度に合った n の値をセットし，outp 関数でこれらの値をこのハードウェアに出力しています．あいだにはいっている io_wait 関数は 100 行目から定義してありますが，これは i8253 などの入出力時間が CPU の命令を実行する速度に追いつかないため，データ落ちを起こさないように，命令と命令が続けて実行される場合あいだに挿入するものです．

```
170: j  = (stops & 0x03)     << 6;
171: j |= (parities & 0x03) << 4;
172: j |= (length & 0x03)   << 2;
173: j |= 0x02;
174:
175: for(i = 0 ; i < 6 ; ++i)
176: {
177: io_wait();
178: outp(USART_COM,0);
179: }
180:
181: io_wait();
182: outp(USART_COM,USART_RESET);
183: io_wait();
184: outp(USART_COM,0x00FF & j);
185: io_wait();
186: outp(USART_COM,0x37);
187: io_wait();
188: inp(USART_DAT);
189: io_wait();
190: inp(USART_DAT);
191:
192: io_wait();
193: o_ppi = inp(PPI_R_C);
194: io_wait();
195: outp(PPI_R_C,(o_ppi & 0xFE));
196: io_wait();
197: o_imr = inp(PIC_M_CW);
198:
199: io_wait();
200: outp(PIC_M_CW,o_imr & 0xEF);
201: io_wait();
202: outp(PPI_R_C,(o_ppi | 1));
203:
```

```
204: /*  Ready to setup the hardware */
205:
206: segread(&segreg);
207: inpreg.h.ah = GET_INT;
208: inpreg.h.al = INT_8251;
209: int86x(INT_DOS,&inpreg,&outreg,&segreg);
210: segread(&segreg);
211: o_seg = segreg.es;
212: o_off = outreg.x.bx;
213:
214: cli();
215: inpreg.h.ah = SET_INT;
216: inpreg.h.al = INT_8251;
217: segreg.ds    = (unsigned)((long)rsintr >> 16);
218: inpreg.x.dx = (unsigned)rsintr;
219: int86x(INT_DOS,&inpreg,&outreg,&segreg);
220: cli();
```

リスト5-6　SIO.C②

170行目から173行目ではi8251(シリアルコントローラ)に送るデータを準備します．見てわかるとおり，1バイトのデータのなかにビット単位でこれらの情報をつめこんでやらなければなりません．次ページの**図5-9**にi8251のデータシートを示します．

こういったハードウェアのデータシートは，大きく分けて，

・ICチップそのものに関するもの
・ICチップを組み合わせたハードウェア・システムに関するもの

の2つがあります．このどちらもないと困るものですから，ハードウェアにかかわるプログラムを組む場合はかならず用意しておきましょう．当然，回路図なども読めなければなりませんから，ハードウェアの知識も必須です．デジタル回路のみの場合であれば，たいした経験も必要なく簡単なものですから，ぜひ勉強することをお勧めします．アナログ回路がからんでくると，回路図のみでは追いきれない要素も多くなるため，ほかの勉強も必要になってきます．やる気のある方は，ぜひチャレンジしてみてください．

175行目から190行目までが，i8251の初期化です．最初に6回0を書いているのは，このハードウェアの仕様です．ここでは，次のことを行っています(くわしくはi8251のマニュアルを参照)．

1. i8251のリセット(182行目)
2. 先に作った値をセット(184行目)
3. i8251のDTRをONにし，送受信ともenableにする(186行目)
4. ごみデータがはいっているので除去(188，190行目)

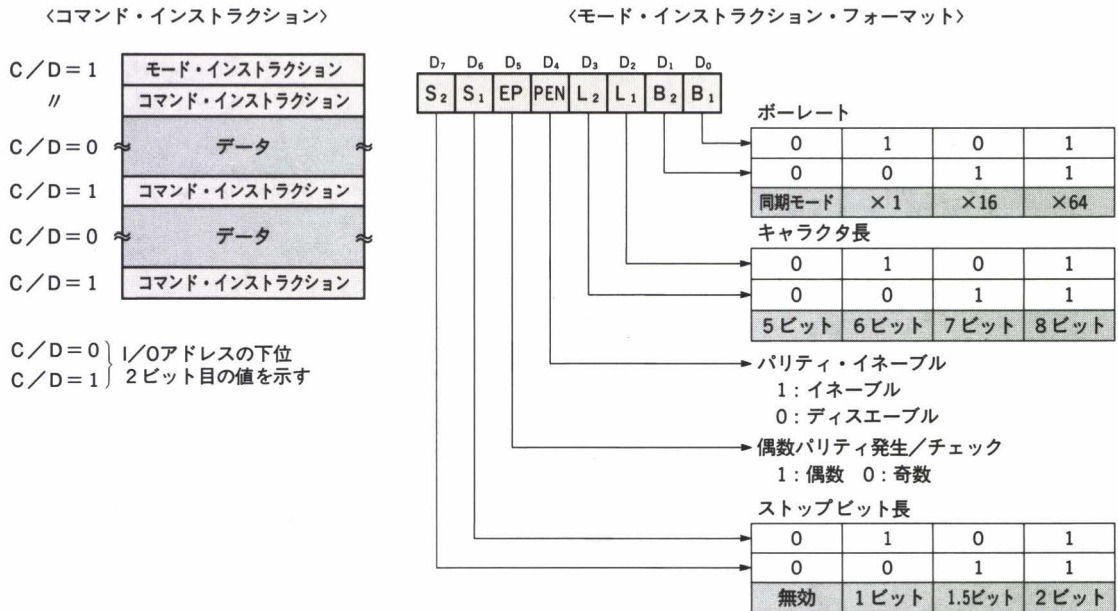



図 5-9 i8251 のデータシート

192 行目から 202 行目は，PC-9801 の割り込み関係のハードウェアのセットアップです．PPI_W_C で示されるハードウェアは i8255 パラレルインターフェイスを使って PC-9801 のハードウェアを制御しているポートで，割り込みの可否を決定します．199 行目から 202 行目では，i8259 割り込みコントローラの「割り込みマスクレジスタ」に，i8251 からの割り込みが許可されるようにそのビットを 0 にしています．前の古いマスクレジスタの値は，o_imr という変数にとっておき，sio_finish 関数で RS-232C が使われなくなったときに，もとへ戻しておきます(**リスト 5-7** の 251 行目)．

206 行目から 212 行目で現在の 8086 系 CPU の RS-232C ハードウェア割り込みのベクタアドレスを取ってきて，やはり sio_finish 関数が呼ばれたときに，もとに戻してやるように準備をしておきます．これは，単に割り込みベクタアドレスを読んでくればよいのですが，MS-DOS のシステムコールにはこういった機能が用意されているのでこれを使っています．

215 行目から 219 行目はこのプログラムで使う割り込みベクタアドレスをセットしています．セットされるベクタアドレスは，割り込みルーチンが始まるラベルのアドレスですから，RSINT.ASM のなかの_rsintr のアドレスをセットします．これも実は単にベクタアドレスを直接書き直してもよいのですが，MS-DOS のシステムコールを使っています．

```
221:
222: /*  Ready to setup the Interrupt vectors     */
223:
224: tail = 0;
225: ptr  = 0;
226: xcont = xc;
227: x_flag = 0;
228:
229: off_size = 3 * (size / 4);
230: on_size  = size / 4;
231:
232: tx_enable = TRUE;
233: t_tail = 0;
234: t_ptr  = 0;
235:
236: sti();
237: return(TRUE);
238: }
239:
240:
241: /* *****************************************************************
242:      finish the sio use
243: ***************************************************************** */
244:
245: int      sio_finish()
246: {
247: cli();
248: io_wait();
249: outp(PPI_R_C,(o_ppi & 0xFE));
250: io_wait();
251: outp(PIC_M_CW,o_imr);
252:
253: segread(&segreg);
254: inpreg.h.ah = SET_INT;
255: inpreg.h.al = INT_8251;
256: segreg.ds   = o_seg;
257: inpreg.x.dx = o_off;
258: int86x(INT_DOS,&inpreg,&outreg,&segreg);
259:
260: io_wait();
261: outp(PPI_R_C,o_ppi);
262: sti();
263: free(siobuf);
264: free(txbuf);
265: }
```

リスト5-7　SIO.C ③

sio_init 関数の最後の部分(224 行目から 237 行目)では，このプログラムの内部で使うパラメータの値を初期化しています．

1. tail(受信バッファの書き込み位置を示す)を 0 に設定(224 行目)
2. ptr(受信バッファの読み出し位置を示す)を 0 に設定(225 行目)
3. xcont(フローコントロールのフラグ)のセット(226 行目)
4. x_flag(フローコントロールの内部フラグ)のリセット(227 行目)
5. 受信フローコントロールの XOFF を出すバッファ内データ数を算出(229 行目)
6. 受信フローコントロールの XON を出すバッファ内データ数を算出(230 行目)
7. 送信側フローコントロールのフラグをリセット(232 行目)
8. 送信バッファの書き込み，読み出し位置を 0 にリセット(233，234 行目)
9. CPU の割り込みを許可(236 行目)

これで sio_init 関数での割り込みサービスルーチンのセットアップを終わります．このあとは，割り込みがかかったときに実質的な働きをする rsintc 関数や，システム環境のセットアップやバッファとのやりとりを行う sio_stat 関数，sio_receive 関数，sio_s_stat 関数，sio_send 関数が使えるようになります．

割り込みベクタなどは，このプログラムが終了したあと，もとの値に戻しておかないと暴走の原因になります．そこで，プログラムの終了時には，245 行目からの sio_finish 関数を呼び出して，これらの値をもとへ戻します．sio_finish 関数では，RS-232C の割り込みを禁止し，RS-232C の割り込みベクタアドレスをもとへ戻し，割りつけてあった受信割り込みバッファと送信バッファをシステムに返しメモリを解放します．

5.5 サンプルプログラム　— VTE —

■ プログラムの概要

ターミナルエミュレータVTEは，C言語のソースファイルが3つ，アセンブラで記述されたファイルが1つ，それ以外のヘッダーファイルが2つという構成になっています．また，これとは別にコンパイル，アセンブル，リンクのためのMAKEFILEファイル(MS-DOS Ver3.1以降のMAKEコマンド用)，リンクのために「RES」という拡張子のつくファイルが用意されています．

SIO.C

RS-232Cインターフェイスの初期化や，受信時の割り込み処理の中身，送信の処理などのほとんどがこのファイルで記述されています．

PUTVT.C

このターミナルエミュレータは，DEC社の専用ターミナルであるVT-100というターミナルと一部が同じ仕様になっています．これは第2章の65ページでも取り上げたように，「0x1B」(エスケープコード)から始まる一連の数バイトのコード(エスケープシーケンス)によって，たとえば「画面をクリアする」，「カーソルを任意の位置に置く」といった文字以外の画面制御を実現するものです．これによって，パソコン通信上でのゲームや，スクリーンエディタなどの複雑な画面の動きを必要とするプログラムが実行できるわけです．

VTE.C

このターミナルプログラムの本体です．ここでは入力されたコマンド行の解釈なども行っています．VTE.CはSIO.CやPUTVT.Cで使われている関数を呼び出すため，そのなかで使われている関数が定義されているSIO.H/PUTVT.Hヘッダーファイルをインクルードします．

RSINT.ASM

前節でも解説したように，このプログラムでは，CPUのレジスタの退避を行い，C言語の関数を呼び出し，それが終わったら退避したすべてのレジスタをもとへ戻しています．

その他

MAKEFILEは，MS-DOS Ver3.1以降についてくるMAKEコマンドによってコンパイルを行うためのファイルです．MAKEについては第7章を参照してください．VTE.RESは，コンパイルされたオブジェクトファイルのすべてとライブラリをリンクし，実行プログラムを作るためにリンカで読まれる「応答ファイル」です．

■ main関数の概要

プログラム全体の基本構成は，main関数を見ればわかります．main関数はVTE.Cの92行目から記述されていますが，最初に画面の初期化を行い(init_scrn関数)，引数の評価を行って(set_sio関数)から，SIO.Cへ渡すの各種パラメータをセットしています．このパラメータにはmain関数の引数である「argc, argv, envp」の3つをそのまま関数に渡しています．

3つ目の引数envpは，システムで使われている環境変数を取り込むときに使います．ここには，環境を表す文字列がargvとまったく同じ形式ではいっていますが，その個数は，文字配列のポインタの配列のNULLで判断することになります．

120行目からのwhile文は131行目まで続き，**図5-10**のループを回っています．

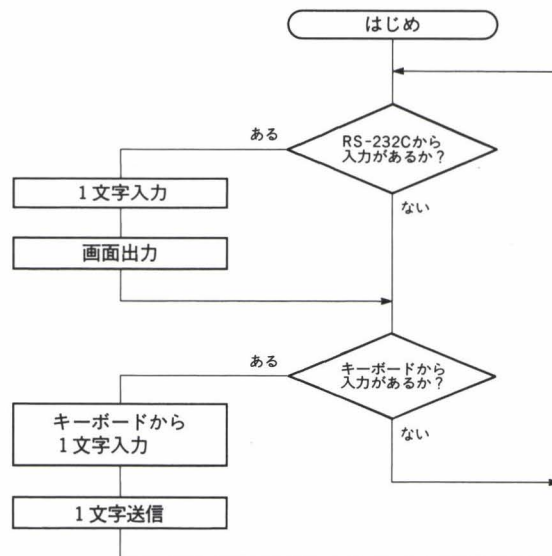



図5-10 メインルーチンのループ

このルーチンのうち，RS-232Cの1文字の受信は，最初の判断で「もし文字が入力されていれば」実行されるわけですが，この判断ルーチンはバッファ(SIO.Cで定義されているsiobuf[　])にRS-232Cから文字が来たかどうかを判断しているだけです．そして，実際の受信もそのバッファから文字をとってくるだけで，ハードウェアを直接アクセスしているわけではありません．ハードウェアのアクセスは割り込みで起動されるルーチン(SIO.Cで定義されているrsintc関数)で行われており，main関数のプログラムではいっさい気にする必要はありません．同様に，キーボードからの1文字入力も，キーボードバッファに文字がはいっているかどうかをみているだけで，キーボードのハードウェアの直接アクセスは行っていません．このような「ハードウェア割り込みによる間接アクセス」のメカニズムは，次ページの**図5-11**に示すように**リングバッファ**という方式をとっています．

このプログラムでは文字の送信にも，まったく同じリングバッファ方式が使われています．この方法によると，非常に高速な送信が可能になり，XON／XOFFのフロー制御が行われている場合なども，キーボードから打った文字をとりこぼすことがありません．RS-232Cへのデータ送信は2つのルーチンsio_send関数，sio_s_stat関数で行っており，どちらもこのmain関数のwhileループ中で呼ばれます．

129行目では，CTRL+Zの文字がキーボードから入ると，ループを抜けるようにしてあります．そして，sio_finish関数(SIO.C)でセットアップしてあったバッファを解放し，このプログラムを終わります．

■ 送受信関数の概要

RS-232Cが上記のsio_init関数でセットアップされると，送受信が行えるわけですが，ここではそこで使われる関数の概略を説明します．

sio_send 関数(272行目)

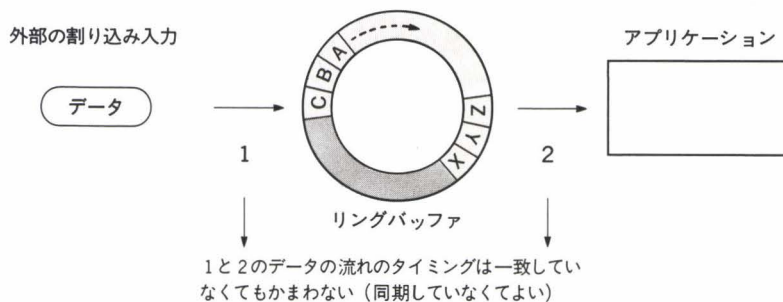
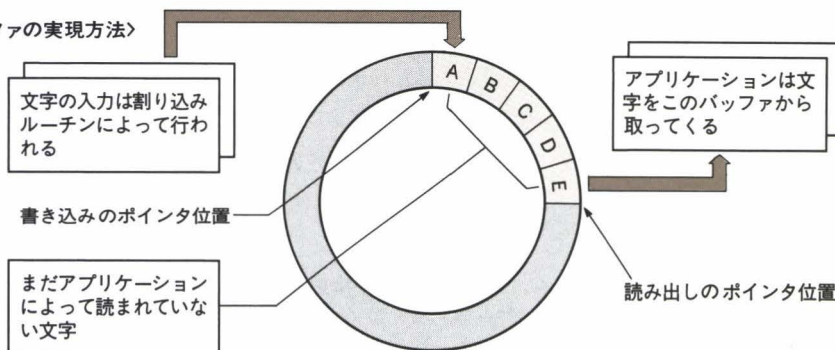
1文字のデータをRS-232Cのバッファに書き込みます．バッファが一杯になったら，またバッファの最初に戻って上書きします．

sio_stat 関数(296行目)

割り込み受信バッファを見て，受信データの有無を調べます．また，受信ハードウェアのエラーの検出とリカバー，さらにはフローコントロールでXOFF中の場合，それを解除し，XONの状態に戻すかどうかを決めます．

sio_s_stat 関数(331行目)

sio_send関数で送信バッファ上に書かれたデータを実際にi8251に送り，データの送信を行います．一度この関数が呼ばれると，バッファが空になるまで送信します．

〈リングバッファの構造と概念〉

〈リングバッファの実現方法〉

〈プログラムでの実現方法〉

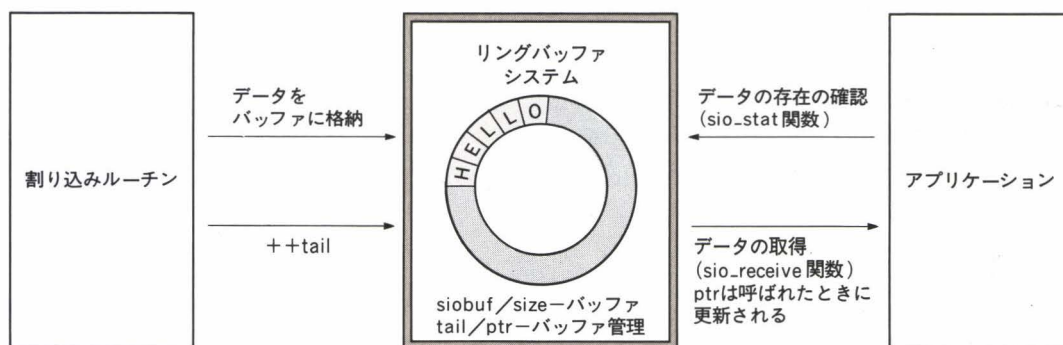



図5-11 リングバッファの概念

sio_receive 関数（365行目）

受信バッファの文字データを1文字ずつ読んできます．データがない場合は0を返しますが，0は正規のデータのときもありますから，アプリケーションではsio_stat関数でデータの有無を判断したあと，この関数を呼ぶかどうかを判断しなればなりません．

sio_break 関数（381行目）

break信号を送信します．

rsintc 関数（396行目）

この関数がアセンブラで記述された割り込みルーチンから呼ばれる〝本当の〟割り込み処理ルーチンです．データをi8251から読み込んで受信バッファにストアし，XOFFのタイミングであればXOFFを送信し，受信した文字データがXONかXOFFであれば，送信のXOFFフラグをONにします．処理が終わったら，またアセンブラで書かれた_rsintrに戻ります．

この関数をデバッグする場合は，割り込みに関係するライブラリ関数を呼んだりしてはいけません．たとえば，デバッグのためにprintf関数を使うことはできません．

sio_enable 関数（452行目）

現在のハードウェアのステータスを読み，送受信可能な状態かどうかを判断します．

ここで紹介したSIO.CとRSINT.ASMは汎用性のあるプログラムですから，ほかのターミナルプログラムやパソコン通信を行うプログラムを作る場合でも，そのままこのルーチンを使うことが可能です．

■ 画面制御関数の概要

PUTVT.Cは数個の関数からなっていますが，ほかのルーチンから呼び出されるのは，putch_vt関数のみで，そのほかの関数はすべてこのputch_vt関数から呼ばれます．

get_csr 関数（38行目）

この関数は，MS-DOSのエスケープシーケンスを使って現在のカーソル位置を知るためのものです．MS-DOSのエスケープシーケンスでは「ESC，'['，'6'，'n'」を送ると，標準入力から現在のカーソル位置をある一定のフォーマットで返してきます（66ページの**表2-3**を参照）．この出力を取ってきて現在のカーソル位置を知るわけです．

putch_vt 関数(82 行目)

この関数は外部から呼ばれますが，その内容は単にフラグの処理をし，必要な関数を呼び出しているだけです．

esc_sup 関数(118 行目)

エスケープシーケンスを認識する関数です．エスケープシーケンスの認識が行われると，実際の解釈と実行はこのあとの seq_valid 関数で行われます．

seq_valid 関数(143 行目)

エスケープシーケンスを解釈し，実行する関数です．

以上で関数ごとの説明は終わります．くわしい処理については，ソースリストをじっくりと見てください．

```
 1: RSINT.OBJ:  RSINT.ASM
 2:     MASM /MX RSINT.ASM;
 3:
 4: SIO.OBJ:    SIO.C
 5:     CL -AL -Gs -Ox -Zi -G0 -W1 -J -c SIO.C
 6:
 7: PUTVT.OBJ:  PUTVT.C
 8:     CL -AL -Gs -Ox -Zi -G0 -W1 -J -c PUTVT.C
 9:
10: VTE.OBJ:    VTE.C
11:     CL -AL -Gs -Ox -Zi -G0 -W1 -J -c VTE.C
12:
13: VTE.EXE:    VTE.OBJ PUTVT.OBJ SIO.OBJ RSINT.OBJ
14:     LINK @VTE.RES
```

リスト 5-8 MAKEFILE

```
1: VTE+
2: PUTVT+
3: SIO+
4: RSINT
5: VTE.EXE /NOD/LINENUMBERS/MAP
6: VTE.MAP
7: LLIBCE
```

リスト 5-9 VTE.RES

```
 1: /*
 2:      Define Values for SIO
 3: */
 4:
 5: #define     DR          0x0080
 6: #define     XF          0x0040
 7: #define     FE          0x0020
 8: #define     OE          0x0010
 9: #define     PE          0x0008
10: #define     CI          0x0004
11: #define     CS          0x0002
12: #define     CD          0x0001
13:
14: #define     B_75        0
15: #define     B_150       1
16: #define     B_300       2
17: #define     B_600       3
18: #define     B_1200      4
19: #define     B_2400      5
20: #define     B_4800      6
21: #define     B_9600      7
22: #define     B_19200     8
23: #define     B_38400     9
24:
25: #define     L_5         0
26: #define     L_6         1
27: #define     L_7         2
28: #define     L_8         3
29:
30: #define     P_NONE      0
31: #define     P_ODD       1
32: #define     P_EVEN      3
33:
34: #define     S_10        1
35: #define     S_15        2
36: #define     S_20        3
37:
38: #define     X_COFF      0
39: #define     X_CON       1
40:
41:
42: extern  void    io_wait();
43: extern  int     set_bufsiz();
44: extern  int     sio_init();
45: extern  int     sio_finish();
46: extern  int     sio_send();
47: extern  int     sio_stat();
48: extern  void    sio_s_stat();
49: extern  int     sio_receive();
50: extern  int     sio_break();
51: extern  void    rsintc();
52: extern  int     sio_h_stat();
```

（注記：5〜12行 信号線のビットのテスト用の値／14〜23行 ボーレート／25〜28行 ビット長／30〜32行 パリティ／34〜36行 ストップビット／38〜39行 XON/XOFF）

リスト5-10　SIO.H

```
1: extern  void    putch_vt();
2: extern  void    esc_sup();
3: extern  int     seq_valid();
```

リスト5-11 PUTVT.H

```
 1: /*
 2:      RS-232C LINE SUPPORTS FOR ASYNCHRONUS COMMUNICATIONS
 3:
 4:      for The Microsoft C-Compiler Ver 5.1 or later.
 5:      for The PC-9801 Series Personal-Computers or Compatibles.
 6:
 7:      CAUTION : This program depends on real HARDWARE's of
 8:                  PC-9801(NEC) or Compatibles.
 9: */
10:
11: #include     <io.h>
12: #include     <dos.h>
13: #include     <string.h>
14: #include     <malloc.h>
15: #include     <setjmp.h>
16: #include     <conio.h>
17:
18: #define      BOOL              int
19: #define      FALSE             0        Bool型の定義
20: #define      TRUE              1
21:
22: #define      SYS_CNT_SET       0x75 ……i8253のカウンタのセット
23: #define      SYS_CNT_MOD       0x77 ……i8253のカウンタのモードレジスタ
24: #define      USART_COM         0x32 ……i8251のコマンドステータスレジスタ
25: #define      USART_DAT         0x30 ……i8251のデータレジスタ
26: #define      PPI_W_C           0x37 ……i8255のポートCへの書き込みアドレス
27: #define      PPI_R_C           0x35 ……i8255のポートCからの読み込みアドレス
28: #define      PPI_R_B           0x33 ……i8255のポートBからの読み込みアドレス
29: #define      PIC_M_CW          0x02 ……i8259のコマンドワード
30:
31: #define      USART_TXE         0x01 ……i8251のTxEビット
32: #define      USART_RXE         0x04 ……i8251のRxEビット
33: #define      USART_RESET       0x40 ……i8251のリセットコマンド
34: #define      INT_DOS           0x21 ……MS-DOSのシステムコール
35: #define      GET_INT           0x35 ……ベクタアドレスを読み込むシステムコール
36: #define      SET_INT           0x25 ……ベクタアドレスをセットするシステムコール
37: #define      INT_8251          0x0C ……i8251のインタラプトベクタ値
38:
39: #define      TCONST            5000 ……タイムアウトの定数
40: #define      TXRDY             0x04 ……i8251のTxReadyビット
41: #define      ERBIT             0x38 ……i8251のエラービット
42: #define      ERRESET           0x37 ……i8251のリセット
43:
44: #define      X_ON              0x11 ……XONのコード
45: #define      X_OFF             0x13 ……XOFFのコード
46: #define      ESC               0x1B ……エスケープシーケンス
```

```
47:
48: #define      NULL              0
49: #define      CPUCLK            8 ……………………CPUのクロックはとりあえず8MHzにセット
50: #define      SPS               2000 ……………スペースカウント値
51: #define      TIMEOUT           (-2) ……………タイムアウトを示すフラグ
52:
53: char     dummy[20];
54:
55: int      speed_tbl[2][10] = ……………………分周比のテーブル
56:          {2048,1024, 512, 256, 128,  64,  32,  16,   8,   4,
57:           1664, 932, 416, 208, 104,  52,  26,  13,   6,   3};
58:
59: int      o_ppi = 0; ……………………PPIインタラプトマスクの初期値
60: int      o_imr = 0; ……………………i8259のIMRの初期値
61: int      o_seg = 0; ……………………i8251インタラプトベクタのオフセットの初期値
62: int      o_off = 0;
63:
64: unsigned int    tail  = 0; ┐
65: unsigned int    ptr   = 0; ┘リングバッファの読み出し／書き込みポインタの初期値
66: unsigned int    size  = 0; ……………リングバッファのサイズの初期値
67:
68: unsigned int    off_size = 0; ┐
69: unsigned int    on_size  = 0; ┘……………XON/XOFFのタイミング
70:
71: int      xcont    = 0;
72: int      x_flag   = 0;
73: char     r_data   = 0; ……………………データバッファのサイズ
74:
75: int      tx_enable = TRUE; ……………TxEnableとフラグをON
76:
77: char    *p; ……………FARコール用の定数
78:
79: char    *siobuf; ……………SIOリングバッファのラベル
80:
81: char    *txbuf; ……………SIO Txバッファのラベル
82:
83: unsigned int    txbsiz; ……………………Txバッファのサイズ
84: unsigned int    t_tail; ……………………Txバッファのtailポインタ
85: unsigned int    t_ptr;  ……………………Txバッファのreadポインタ
86:
87: extern   void  rsintr(); ┐
88: extern   void   cli();   │アセンブラルーチンにある関数
89: extern   void   sti();   ┘
90:
91: union    REGS    inpreg; ┐
92: union    REGS    outreg; │int86xを使うための構造体
93: struct   SREGS   segreg; ┘
94:
95:
```

```
 96: /* *****************************************************************
 97:     time wait functions for OUTP() & INP()
 98: ***************************************************************** */
 99:
100: void     io_wait()……………………I/O時のウェイト用関数(ダミー)
101: {
102: int      i;
103: ;
104: }
105:
106:
107: /* *****************************************************************
108:     Get SIO-Buffer Dynamically
109: ***************************************************************** */
110:
111: BOOL     set_bufsiz()
112: {
113: size = 0xFFF0;……………バッファサイズの最大は65520バイト
114: while((char *)NULL == (siobuf = (char *)malloc(size)))
115:     {
116:     size /= 2;
117:     if(0x100 > size) return(FALSE);
118:                                 /* Not enough memory Space */
119:     }
120:
121: if((char *)NULL == (siobuf = (char *)realloc(siobuf,size / 2)))
122:     return(FALSE);
123:                         └→バッファサイズを半分にして
                              reallocする
124: txbsiz = 0xFFF0;
125: while((char *)NULL == (txbuf = (char *)malloc(txbsiz)))
126:     {
127:     txbsiz /= 2;
128:     if(0x100 > txbsiz) return(FALSE);
129:                                 /* Not enough memory Space */
130:     }
131:
132: if((char *)NULL == (txbuf = (char *)realloc(txbuf,txbsiz / 2)))
133:     return(FALSE);
134:
135: size   /= 2;
136: txbsiz /= 2;
137:
138: return(TRUE);
139: }
140:
141:
142: /* *****************************************************************
143:     Initial SIO
144: ***************************************************************** */
145:
146: int      sio_init(int speed, int length, int parities, int stops,
    int xc)
147:     /* speed    : Baud-rates     /[0-9] for [75bps-38400bps] */
148:     /* length   : Data-bit length/[0-3] for [5bits-8bits] */
149:     /* parities : Parity Bits    /[0,2=NO / 1=Odd / 3=Even] */
150:     /* stops    : Stop Bits      /[1=1bit / 2=1.5bits / 3=2bits] */
151:     /* xc       : x-flow control /[0=OFF / 1=ON] */
```

（113〜122行の注記）割り込み受信バッファの領域を確保

（124〜133行の注記）送信バッファの領域を確保

```
152: {
153: int      i, j;
154:
155: if(!set_bufsiz()) return(FALSE);
156: cli();
157: p = (char *)0x00000501L; ……………………CPUクロックの判断(PC-9801シリーズの場合)
158: if((0x10 == (0x70 & *p)) || (0 == (0x80 & *p)))  ┐
159:      i = 0;                                        │
160: else                                               ├ クロック分周テーブルの選択
161:      i = 1;                                        ┘
162:
163: io_wait();                                                   ┐
164: outp(SYS_CNT_MOD,0xB6); …………カウンタ値の再ロード             │
165: io_wait();                                                   │
166: outp(SYS_CNT_SET,speed_tbl[i][speed]);        ┐ データの     ├ i8253の設定
167: io_wait();                                     │ 読み込み     │
168: outp(SYS_CNT_SET,speed_tbl[i][speed] >> 8);   ┘              ┘
169:
170: j  = (stops & 0x03)     << 6;  ┐
171: j |= (parities & 0x03) << 4;   ├ i8251のコマンドワードの作成
172: j |= (length & 0x03)    << 2;  ┘
173: j |= 0x02;
174:
175: for(i = 0 ; i < 6 ; ++i)
176: {
177: io_wait();
178: outp(USART_COM,0); ……………i8251のリセット
179: }
180:
181: io_wait();
182: outp(USART_COM,USART_RESET); ……………………i8251のリセット
183: io_wait();
184: outp(USART_COM,0x00FF & j); ……………………コマンドワードのセット
185: io_wait();
186: outp(USART_COM,0x37); ……………………モードワードのセット
187: io_wait();
188: inp(USART_DAT);  ┐
189: io_wait();       ├ ダミーデータの読み込み
190: inp(USART_DAT);  ┘
191:
192: io_wait();                                                      ┐
193: o_ppi = inp(PPI_R_C); ……………PPI_R_Cの値を読み込む               │
194: io_wait();                                                      │
195: outp(PPI_R_C,(o_ppi & 0xFE)); …………LSBをリセット                │
196: io_wait();                                                      │
197: o_imr = inp(PIC_M_CW); …………………i8259のIMRの値を               │ システムポートi8255と
198:                                     読み込む                     ├ 割り込みコントローラ
199: io_wait();                                                      │ i8259の設定
200: outp(PIC_M_CW,o_imr & 0xEF); ……………i8259にIMR                 │
201: io_wait();                                 の値をセット          │
202: outp(PPI_R_C,(o_ppi | 1)); ………PPI_R_Cに値を書く(受信のハー     │
203:                                   ドウェア割り込みを許可する)     ┘
204: /*  Ready to setup the hardware */
205:
```

```
206: segread(&segreg);
207: inpreg.h.ah = GET_INT;
208: inpreg.h.al = INT_8251;
209: int86x(INT_DOS,&inpreg,&outreg,&segreg);
210: segread(&segreg);
211: o_seg = segreg.es;
212: o_off = outreg.x.bx;
213:
214: cli();
215: inpreg.h.ah = SET_INT;
216: inpreg.h.al = INT_8251;
217: segreg.ds    = (unsigned)((long)rsintr >> 16);
218: inpreg.x.dx = (unsigned)rsintr;…………アセンブラルーチンの
219: int86x(INT_DOS,&inpreg,&outreg,&segreg);  オフセットのセット
220: cli();
221:
222: /*  Ready to setup the Interrupt vectors     */
223:
224: tail = 0;
225: ptr  = 0;
226: xcont = xc;
227: x_flag = 0;
228:
229: off_size = 3 * (size / 4);
230: on_size  = size / 4;
231:
232: tx_enable = TRUE;
233: t_tail = 0;
234: t_ptr  = 0;
235:
236: sti();
237: return(TRUE);
238: }
239:
240:
241: /* ******************************************************************
242:     finish the sio use
243: ****************************************************************** */
244:
245: int      sio_finish()………………………i8251のベクタ値の復帰と終了の処理
246: {
247: cli();
248: io_wait();
249: outp(PPI_R_C,(o_ppi & 0xFE));……………………PPIの割り込みの禁止
250: io_wait();
251: outp(PIC_M_CW,o_imr);………………………PICのコマンドワークをもとに戻す
252:
253: segread(&segreg);
254: inpreg.h.ah = SET_INT;
255: inpreg.h.al = INT_8251;
256: segreg.ds    = o_seg;
257: inpreg.x.dx = o_off;
258: int86x(INT_DOS,&inpreg,&outreg,&segreg);
259:
260: io_wait();
261: outp(PPI_R_C,o_ppi);………………………PPIの値をもとに戻す
262: sti();………………………割り込みの許可
```

206〜212: 現在のインタラプトベクタの値を読み込みセーブしておく

217: アセンブラルーチンのセグメントのセット

214〜220: インタラプトベクタ値を書き換える

229〜230: XONとXOFFのタイミングの計算

224〜234: 各種変数の初期化

253〜258: インタラプトベクタをもとに戻す

```
263: free(siobuf);  } メモリの開放
264: free(txbuf);   }
265: }
266:
267:
268: /* ****************************************************************
269:      Send 1-char to Tx-buffer
270: **************************************************************** */
271:
272: int      sio_send(int c) ························Txバッファに1文字送る
273: {
274: unsigned int     t_len;
275:
276: if(t_tail >= t_ptr)
277:      t_len = t_tail - t_ptr; ························Txバッファにたまっている文字列の長さ
278: else
279:      t_len = t_tail + txbsiz - t_ptr;
280:                                        /* If buffer is full */
281: if(t_len >= txbsiz)
282:      return(FALSE);
283:
284: txbuf[t_tail] = (char)c;
285: ++t_tail;
286: if(t_tail >= txbsiz)  } tailポインタの調整
287:      t_tail = 0;      }
288: return(TRUE);
289: }
290:
291:
292: /* ****************************************************************
293:      receive buffer check
294: **************************************************************** */
295:
296: BOOL sio_stat() ························受信バッファのチェック
297: {
298: unsigned int     len;
299:
300: if(t_tail >= t_ptr)
301:      len = t_tail - t_ptr;
302: else
303:      len = t_tail + txbsiz - t_ptr;
304:
305: if((len < on_size) && x_flag) ····················XOFFのタイミングの検出
306:      {
307:      x_flag = FALSE; ························Xflagのリセット
308:      while(0 == (TXRDY & inp(USART_COM))); ····················XON送出
309:      outp(USART_DAT,X_ON);
310:      }
311:
312: if(0 != (ERBIT & inp(USART_COM))) ························エラーが起きた場合の処理
313:      {
314:      cli();
315:      outp(USART_COM,ERRESET); ····················エラーステータスをリセット
316:      sti();
317:      return(FALSE);
318:      }
319:
```

```
320: if(tail == ptr) ………………………データがない場合
321:     return(FALSE);
322: return(TRUE);
323: }
324:
325:
326: /* ******************************************************************
327:     This routine called by Main-loop
328:     to flush the Send sio buffer
329: ****************************************************************** */
330:
331: void    sio_s_stat() …………………送信バッファから送信ハードウェアへデータを送出
332: {
333: unsigned int    len;
334: long    i;
335:
336: while(TRUE)
337:     {
338:     if(t_tail >= t_ptr)
339:         len = t_tail - t_ptr;
340:     else
341:         len = t_tail + txbsiz - t_ptr;
342:
343:     if(len == 0) ………………………データがない場合
344:         return;
345:     if(!xcont) ……………XCONTがOFFの場合
346:         tx_enable = TRUE;
347:     if(!tx_enable) ………………………送信ステータスがenableかどうか見ている
348:         return;
349:     if(0 != (0x0040 & inp(PPI_R_B))) ………………CSがUPするまで待つ
350:         return;
351:     while(0 == (TXRDY & inp(USART_COM))); ……………Sendステータスが立つまで待つ
352:
353:     outp(USART_DAT,txbuf[t_ptr]); …………………データを１バイト送信
354:     ++t_ptr;                  ┐
355:     if(t_ptr >= txbsiz)       ├ バッファ・ポインタの調整
356:         t_ptr = 0;            ┘
357:     }
358: }
359:
360:
361: /* ******************************************************************
362:     Read 1-character from receive buffer
363: ****************************************************************** */
364:
365: int     sio_receive() ………………………受信バッファからアプリケーションへ
366: {                                  データを送出
367: if(ptr == tail)
368:     return(0);
369: r_data = 0x00FF & (int)siobuf[ptr];
370: ptr++;
371: if(ptr >= size)
372:     ptr = 0;
373: return(r_data);
374: }
375:
376:
```

```
377: /* ******************************************************************
378:      Send BREAK
379: ****************************************************************** */
380:
381: int      sio_break()……………………ブレーク信号の送出
382: {
383: long     i;
384:
385: outp(USART_COM,0x3F);
386: for(i = 0 ; i < 50000 ; ++i);
387: outp(USART_COM,0x37);
388: }
389:
390:
391: /* ******************************************************************
392:      Interrupt Support Routines
393:      Called by RSINT.ASM : RSINTR Routines
394: ****************************************************************** */
395:
396: void     rsintc()……………………受信割り込みの処理で，アセンブラルーチン
397: {                               から呼ばれる関数
398: unsigned int     len;
399: char         ct;
400:
401: ct = (char)inp(USART_DAT);………………送られてきた文字を読み込む
402:
403: if((ct == X_OFF) && xcont && tx_enable)  ┐
404:      {                                    │
405:      tx_enable = FALSE;                   ├ XOFF時の処理
406:      return;                              │
407:      }                                    ┘
408:
409: if((ct == X_ON) && xcont && (!tx_enable)) ┐
410:      {                                     │
411:      tx_enable = TRUE;                     ├ XON時の処理
412:      return;                               │
413:      }                                     ┘
414:
415: if(tail >= ptr)                  ┐
416:      len = tail - ptr;           │
417: else                             ├ リングバッファのサイズの計算
418:      len = tail + size - ptr;    ┘
419: if(len >= size) ┐
420:      return;     ┘ バッファがいっぱいの場合
421:
422: siobuf[tail] = ct;…………………バッファに文字をストア
423: tail++;         ┐
424: if(tail >= size)├ ポインタの調整
425:      tail = 0;   ┘
426:
427: if((0 == xcont) && x_flag) ┐
428:      {                      │
429:      x_flag = FALSE;        ├ XOFFの場合，リターン
430:      return;                │
431:      }                      ┘
432:
```

```
433: if(tail >= ptr)
434:     len = tail - ptr;           バッファ・ポインタの調整
435: else
436:     len = tail + size - ptr;
437:
438: if((len > off_size) && (!x_flag)) ………………XOFF時の処理
439:     {
440:     x_flag = TRUE; ………………x_flagのセット
441:     while(0 == (TXRDY & inp(USART_COM)));
442:     outp(USART_DAT,X_OFF); ………………XOFFを送る
443:     return;
444:     }
445: }
446:
447:
448: /* ***************************************************************
449:     Line is enable or not ?
450: *************************************************************** */
451:
452: BOOL    sio_enable()
453: {
454: int     c;
455:
456: if(0 == (0x0080 & inp(USART_COM)))   DSRを読む
457:     return(FALSE);
458: if(0 != (0x0060 & inp(PPI_R_B)))
459:     return(FALSE);                   CSとCDを読む
460: return(TRUE);
461: }
```

リスト5-12 SIO.C

```
 1: /*
 2:     Putch with VT-100 Escape Sequence
 3: */
 4:
 5: #include    <stdio.h>
 6: #include    <io.h>
 7: #include    <conio.h>
 8: #include    <ctype.h>
 9: #include    <string.h>
10: #include    <stdlib.h>
11:
12: #define     ESC         0x1B
13:
14: #define     BOOL        int
15: #define     FALSE       0
16: #define     TRUE        1
17: #define     SAME        0
18:
19: #define     FIXSEQ      16 ………………VT-100エミュレートのシーケンス数
20:
21: static  int     esc_flag = FALSE; …エスケープシーケンスを退避したかどうかを示すフラグ
22: static  char    seq_buf[20];……………エスケープシーケンスを退避するバッファ
23: static  int     seq_cnt = 0;……………バッファのカウンタ
24:
25: static  char    str_seq0[FIXSEQ][7] = { "D","E","M","*",              ┐ エスケープ
26:                 ")0",")3","[>5l","[>5h","[>1h",                        │ シーケンス
27:                 "[>1l","[>3h","[>3l","[s","[u","[?1h","="};          ┘ のテーブル
28:
29: /*
30:     Get now Cursor-Position
31:
32:     Calling Sequence :    int   x,y;     ----> data is <int>
33:                           get_csr(&x,&y);   ----> call by Address.
34:                           position-X = x (0 to 79)
35:                           position-Y = y (0 to 24)
36: */
37:
38: void    get_csr(int *x, int *y)
39: {                   └→行    └→列
40: int     i,j;
41: char    esc_buf[20];
42: char    esc_l[3];
43: char    esc_c[3];
44:
45: printf("¥033[6n");……………………カーソル位置の取得
46:
47: for(i = 0 ; i < 20 ; ++i)              ┐
48:     {                                  │
49:     esc_buf[i] = getch();              │ カーソル位置の退避
50:     if(esc_buf[i] == 'R') break;       ┘
51:     }
52:
```

```
 53: esc_buf[i] = 0;
 54:
 55: i = 2;
 56: j = 0;
 57: while(esc_buf[i] != ';')
 58:     {
 59:     esc_l[j] = esc_buf[i];          行位置の取得
 60:     j++;i++;
 61:     }
 62:
 63: esc_l[j] = 0;
 64: j = 0; i++;
 65:
 66: while(esc_buf[i] != 0)
 67:     {
 68:     esc_c[j] = esc_buf[i];          列位置の取得
 69:     j++; i++;
 70:     }
 71:
 72: esc_c[j] = 0;
 73: *x = atoi(esc_c) - 1;
 74: *y = atoi(esc_l) - 1;
 75: }
 76:
 77:
 78: /*
 79:     Character Entry like putch() Sequence
 80: */
 81:
 82: void    putch_vt(int c)
 83: {
 84: int     flg_tmp;
 85: void    esc_sup();
 86:
 87: c &= 0x00FF; ......................下位ビットにマスクをかける
 88:
 89: flg_tmp  = esc_flag;
 90: flg_tmp |= ((c == ESC) << 1);        エスケープフラグのセット
 91:
 92: switch(flg_tmp)......................エスケープシーケンスか否かの判断
 93:     {
 94:     case    0:
 95:         putch(c);        エスケープシーケンスでない
 96:         return;
 97:     case    1:
 98:         esc_sup(c);        エスケープシーケンスの第nバイト
 99:         return;
100:     case    2:
101:         esc_flag = TRUE;        エスケープシーケンスの第1バイト
102:         return;
103:     case    3:
104:         esc_flag = FALSE; seq_cnt = 0;
105:         seq_buf[0] = 0;                    エスケープシーケンスでない
106:         putch(c);
107:         return;
108:     default:
109:         return;
```

```
110:        }
111: }
112:
113:
114: /*
115:       Escape Sequence Support
116: */
117:
118: void     esc_sup(int c)
119: {
120: BOOL     seq_valid();
121:
122: if(seq_cnt >= 19)……………………エスケープシーケンスが多すぎる場合
123:      {
124:      seq_cnt = 0;
125:      seq_buf[0] = 0;
126:      esc_flag = FALSE;
127:      return;
128:      }
129:
130: seq_buf[seq_cnt] = c; ……………………バッファにストア
131: seq_cnt++;
132: seq_buf[seq_cnt] = 0;
133:
134: if(seq_valid())……………………エスケープシーケンスのテスト
135:      {
136:      seq_cnt = 0;
137:      seq_buf[0] = 0;
138:      esc_flag = FALSE;
139:      return;
140:      }
141: }
142:
143: BOOL     seq_valid()……………………エスケープシーケンスを送出
144: {
145: int      i;
146:
147: for(i = 0 ; i < FIXSEQ ; ++i)
148:      {
149:      if(SAME == strcmp(&str_seq0[i][0],seq_buf))
150:           {
151:           switch(i)
152:                {
153:                int      xx,yy;
154:
155:                case     0:
156:                     fputs("¥033[1B",stdout);
157:                     break;
158:                case     1:
159:                     fputs("¥015¥012",stdout);
160:                     break;
161:                case     2:
162:                     get_csr(&xx,&yy);
163:                     if(0 == yy)
164:                          fputs("¥033[01L",stdout);
```

```
165:                  else
166:                      fputs("¥033[1A",stdout);
167:                  break;
168:              case    3:
169:                  fputs("¥033[2J",stdout);
170:                  break;
171:              case    4:
172:                  break;
173:              case    5:
174:                  break;
175:              case    6:
176:                  break;
177:              case    7:
178:                  break;
179:              case    8:
180:                  break;
181:              case    9:
182:                  break;
183:              case    10:
184:                  break;
185:              case    11:
186:                  break;
187:              case    12:
188:                  fputs("¥033[s",stdout);
189:                  break;
190:              case    13:
191:                  fputs("¥033[u",stdout);
192:                  break;
193:              case    14:
194:                  break;
195:              case    15:
196:                  break;
197:              }
198:          return(TRUE);
199:          }
200:      }
201:
202: if(seq_buf[0] != '[')
203:      {
204:      seq_cnt = 0;
205:      seq_buf[0] = 0;
206:      esc_flag = FALSE;
207:      return(FALSE);
208:      }
209:
210: switch(seq_buf[seq_cnt - 1])
211:      {
212:      case    'H':
213:      case    'f':
214:      case    'A':
215:      case    'B':
216:      case    'C':
217:      case    'D':
218:          putch(ESC);
219:          fputs(seq_buf,stdout);
220:          return(TRUE);
```

```
221:     case    'M':
222:         putch(ESC);
223:         fputs(seq_buf,stdout);
224:         putch('¥r');
225:         return(TRUE);
226:     case    'L':
227:         putch(ESC);
228:         fputs(seq_buf,stdout);
229:         putch('¥r');
230:         return(TRUE);
231:     case    'm':
232:         putch(ESC);
233:         fputs(seq_buf,stdout);
234:         return(TRUE);
235:     case    'K':
236:         putch(ESC);
237:         fputs(seq_buf,stdout);
238:         return(TRUE);
239:     case    'J':
240:         putch(ESC);
241:         fputs(seq_buf,stdout);
242:         return(TRUE);
243:     default:
244:         break;
245:     }
246: return(FALSE);
247: }
```

リスト5-13 PUTVT.C

```
 1: /*
 2:     VT-100 Emulators for NEC PC-9801 Series Personal Computers.
 3: */
 4:
 5: #include    <stdio.h>
 6: #include    <conio.h>
 7: #include    <ctype.h>
 8: #include    <string.h>
 9: #include    <stdlib.h>
10:
11: #include    "sio.h"
12: #include    "putvt.h"
13:
14: #define     DEF_SPEED   B_2400  /* default Speed */
15: #define     DEF_LEN     L_8     /* default Data-Length */
16: #define     DEF_PARI    P_NONE  /* default Parity */
17: #define     DEF_STOP    S_10    /* default Stop-bits */
18: #define     DEF_XCONT   X_CON   /* default flow-control */
19:
20: #define     MAX_COM     5 ························オプションの数
21:
```

```
22: char     para_tbl[MAX_COM][3] = {"-B","-D","-P","-S","-X"};
23:                                   /* for command-line search */
24: char     tbl[50][6]    = {"75"  ,"150" ,"300" ,"600"  ,"1200",
25:                            "2400","4800","9600","19200","38400",
26:                            "5","6","7","8"," "," "," "," "," "," ",
27:                            "N","O","N","E"," "," "," "," "," "," ",
28:                            " "  ,"1"  ,"1.5","2"  ," "  ,
29:                            " "  ," "  ," "  ," "  ," "  ,
30:                            "OFF","ON" ," "  ," "  ," "  ,
31:                            " "  ," "  ," "  ," "  ," "  };
32:
33: int param[MAX_COM] = {DEF_SPEED,DEF_LEN,DEF_PARI,DEF_STOP,DEF_XCONT};
34:
35: #define     SAME        0
36:
37: /*
38:     Translate the strings to Upper-Case all
39: */
40:
41: void    toup_s(char strings[])
42: {
43: int     i;
44:
45: for(i = 0 ; (char)0 != strings[i] ; ++i)
46:     strings[i] = (char)toupper((int)strings[i]);
47: }
48:
49:
50: /*
51:     Set SIO parameters & initialize the SIO
52: */
53:
54: void    setsio(int argc, char **argv, char **envp)
55: {
56: int     i, j, k;
57:
58: for(i = 0 ; i < MAX_COM ; ++i)………………オプションの取得
59:     {
60:     for(j = 1 ; j < argc ; ++j)
61:         {
62:         toup_s(argv[j]);
63:         if(SAME == strcmp(argv[j],&para_tbl[i][0]))
64:             {
65:             if( argv[j+1] == (char *)NULL) break;
66:             if(*argv[j+1] == '-')          continue;
67:             for(k = 0 ; k < 10 ; ++k)…………パラメータ・テーブルのサーチ
68:                 {
69:                 toup_s(argv[j+1]);
70:                 if(SAME == strcmp(argv[j+1],&tbl[10 * i + k][0]))
71:                     {
72:                     param[i] = k;
73:                     break;
74:                     }
75:                 }
76:             }
77:         }
```

```
 78:      }
 79: sio_init(param[0],param[1],param[2],param[3],param[4]); ·····SIOの初期化
 80: }
 81:
 82:
 83: /*
 84:      initial Screen
 85: */
 86:
 87: void     init_scrn()
 88: {
 89: printf("¥n¥n¥033[0m¥033[2J¥033)0¥033[>51¥033[>11¥033[>31");
 90: }
 91:
 92: main(int argc, char **argv, char **envp)
 93: {
 94: int      c_send;
 95: int      c_receive;
 96:
 97: init_scrn();
 98:
 99:                                        /* Write CopyRight */
100: printf("VT-100 Easy Terminal Emulators for PC-9801 Series.¥n");
101: printf("Copyright 1987/02/12 CoreDump Co.,Ltd.¥n");
102: printf("Created by Norihiro Mita all rights reserved.¥n¥n");
103:
104: if(SAME == strncmp("-h",argv[1],2))
105:      {
106:      printf("VTE Terminal-Emulator HELP¥n¥n");
107:      printf("VTE -b [Communication-Speed] from 75 to 38400 specified
     ¥n");
108:      printf("    -d [Data-Length]            from  5 to 8 bits.¥n");
109:      printf("    -p [Parity]                 NONE or ODD or EVEN.¥n");
110:      printf("    -s [Stop-Bits]              1 or 1.5 or 2 bits.¥n");
111:      printf("    -x [X-ON/X-OFF]             ON or OFF.¥n");
112:
113:      exit(0);
114:      }
115:
116: setsio(argc,argv,envp); ························SIOの定義
117:
118: printf("¥n¥n(Control-Z to Quit This procedures.)¥n¥n");
119:
120: while(1)·····················メインループ
121:      {
122:      if(sio_stat())
123:           {
124:           c_receive = sio_receive();
125:           if(c_receive != (char)0) putch_vt(c_receive);
126:           }
127:
128:      if(kbhit())     sio_send(c_send = getch());
129:      if(c_send == ('Z' - ' ' - ' ')) break;··········CTRL+Zでメインループを抜ける
130:      sio_s_stat();························送信バッファをフラッシュする
131:      }
```

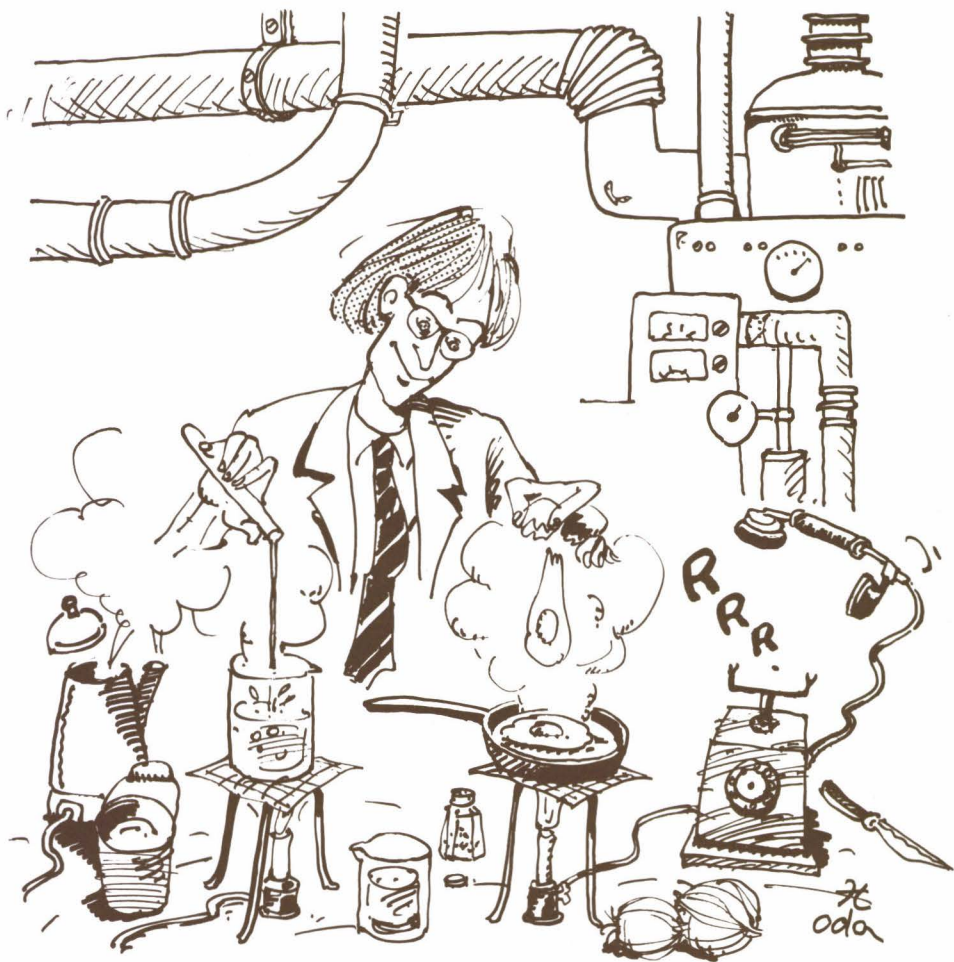
```
132: sio_finish(); ........................後処理を行う
133: printf("¥n¥n¥033[2J");
134: printf("VTE Terminated.¥n");
135: }
```

リスト 5-14 VTE.C

```
  1: ;
  2: ;    Static Name Aliases
  3: ;
  4:       TITLE    RSINT
  5: ;
  6: ;
  7: RSINT_TEXT   SEGMENT BYTE PUBLIC 'CODE'
  8: RSINT_TEXT   ENDS
  9: ;
 10: SIO_TEXT     SEGMENT BYTE PUBLIC 'CODE'
 11: EXTRN        _rsintc:FAR
 12: SIO_TEXT     ENDS
 13: ;
 14: _DATA        SEGMENT WORD PUBLIC 'DATA'
 15: _DATA        ENDS
 16: ;
 17: CONST        SEGMENT WORD PUBLIC 'CONST'
 18: CONST        ENDS
 19: ;
 20: _BSS         SEGMENT WORD PUBLIC 'BSS'
 21: _BSS         ENDS
 22: ;
 23: DGROUP       GROUP   CONST,  _BSS,   _DATA
 24:              ASSUME  CS: RSINT_TEXT, DS: DGROUP, SS: DGROUP,
     ES: DGROUP
 25: ;
 26: RSINT_TEXT   SEGMENT
 27: ;
 28: ;    Interrupt CALLed with RS-232C
 29: ;
 30:              PUBLIC  _rsintr
 31: _rsintr      PROC FAR
 32: ;
 33:              cli ....................処理中は他の割り込みを受けないようにする
 34:              pushf
 35:              push    ax
 36:              push    bx
 37:              push    cx
 38:              push    dx
 39:              push    bp
 40:              mov     bp,sp           すべてのレジスタを退避
 41:              push    si
 42:              push    di
 43:              push    ds
 44:              push    es
 45: ;
```

```
46:             mov     ax,DGROUP                                    DS, ESレジスタ
47:             mov     ds,ax           ; Set DS: to DATA-SEGMENTS   にDGROUPのア
48:             mov     es,ax                                        ドレスをセット
49: ;
50:             cld                     ; Clear Direction Flag
51: ;
52:             call    _rsintc         ; Call C-language Routine……C言語ルーチン
53: ;                                                                のコール
54:             pop     es
55:             pop     ds
56:             pop     di
57:             pop     si
58:             mov     sp,bp
59:             pop     bp
60:             pop     dx      退避したレジスタをもとに戻す
61:             pop     cx
62:             pop     bx
63:             mov     al,20h  ……i8259に対してEOI(End of Interrupt)
64:             out     0,al      を発行し，ハードウェアに割り込み処
65:             pop     ax        理の終了を伝える
66:             popf
67:             sti
68:             iret
69: ;
70: _rsintr     ENDP
71: ;
72: ;       INTERRUPT CLEAR FUNCTIONS
73: ;
74: PUBLIC      _cli        C言語のプログラムからFAR CALLで呼ばれるルーチン
75: _cli        PROC FAR
76: ;
77:             cli……外部割り込みの禁止
78:             ret
79: ;
80: _cli        ENDP
81: ;
82: ;       INTERRUPT SET FUNCTIONS
83: ;
84: PUBLIC      _sti        C言語のプログラムからFAR CALLで呼ばれるルーチン
85: _sti        PROC FAR
86: ;
87:             sti……外部割り込みの許可
88:             ret
89: ;
90: _sti        ENDP
91: ;
92: RSINT_TEXT  ENDS
93: ;
94: END
95: ;
```

リスト5-15　RSINT.ASM

第6章
UNIX上での
プログラミング

C 言語はもともと UNIX オペレーティングシステムを記述するために作られた言語です．したがって，UNIX 上のさまざまなアプリケーションやツール類は，C 言語を使って書かれたものがほとんどです．この章では，UNIX 上でプログラムを組むときに必要なシステムコールについて解説します．

バークレー版(BSD)の UNIX は，大学の研究室などで数多く使用されており，使いやすく完成されたシステムとして定評があります．筆者もこれらの恩恵にあずかった人間ですが，最近ではワークステーションと呼ばれる小型で高性能の UNIX コンピュータが多くなり，UNIX もいよいよ一般的なものとなってきました．しかし，こういったワークステーション上の UNIX の多くは，BSD ではなく AT&T のオリジナルの System V です．また，BSD もこれらオリジナル版 UNIX の機能をとりいれるようになってきており，UNIX System V が数の上では本流となりそうな気配もあります．そこで，この章では BSD 系と System V 系の機能を含んだ UNIX 環境の概念と主なシステムコールを取り上げることにします．

6.1 マルチタスク環境の考え方

今後数年のうちにパソコン上のOSも，これまで主流であったMS-DOSからOS/2への移行が進み，パソコンでもマルチタスクが当然の環境となる日もそう遠くはないでしょう．そこでUNIXだけでなく，より広い意味でのマルチタスク環境の概念とプログラミングの考え方について解説します．

■ シングルタスクとマルチタスク

MS-DOSマシンのような従来のパソコンでは，一度に1つの仕事を行い，またそれしかできないというのが普通でした．しかし，UNIXのようなマルチタスク・オペレーティングシステムを使うと，MS-DOSのようなシングルタスクのオペレーティングシステムとは異なり，複数の仕事を同時に処理できるようになります．またUNIXは，マルチユーザー機能も持っているので，一度にたくさんの人が同じコンピュータ上で違う仕事を行うことが可能です．

これだけであれば，パソコンを複数台並べて使ったのとまったく同じではないかと思われる人も多いでしょう．そこで，こういった1つのCPUがマルチタスクで動くことの利点を以下に挙げてみます．

・同じディスクを複数の人間(もしくはプログラム)が一度に使える

これを「資源(リソース)の共有」といいます．ディスクだけでなく，ほかの外部機器やCPU自身も「資源」として扱われるので，CPUの待ち時間の減少やプリンタの共有化がはかれます．

・複雑な処理をより簡単な処理に分けて記述できる

従来のシングルタスク・オペレーティングシステムでは，通信をしながらワープロをたたくといった処理をするのは，不可能ではないものの非常に困難でした．しかし，マルチタスク処理が行えると，こういった役目の違うプログラムは別々に記述すればよいので，構造が簡単になり，作る方も使う方も快適な環境が得られることになります．

■ マルチタスク環境の実現

マルチタスク・オペレーティングシステムはどのような原理で動いているのでしょうか？　CPUが1つということは，当然ある瞬間には1つの仕事しかできません．そこであるきっかけをハードウェア的，ソフトウェア的に作ってやることで，プログラムを細かい時間単位に分割し，それぞれのタスク(仕事)を少しずつ実行します．この方式を**時分割**(Time Sharing)と呼びます(**図6-1**を参照)．

コンピュータの実行速度は，プリンタへの印字やCRT画面への出力，人間がキーボードから打ち込む速度に比べてかなり速いので，こういった時分割方式が有効になり，CPUの待ち時間を有効に活用することができます．

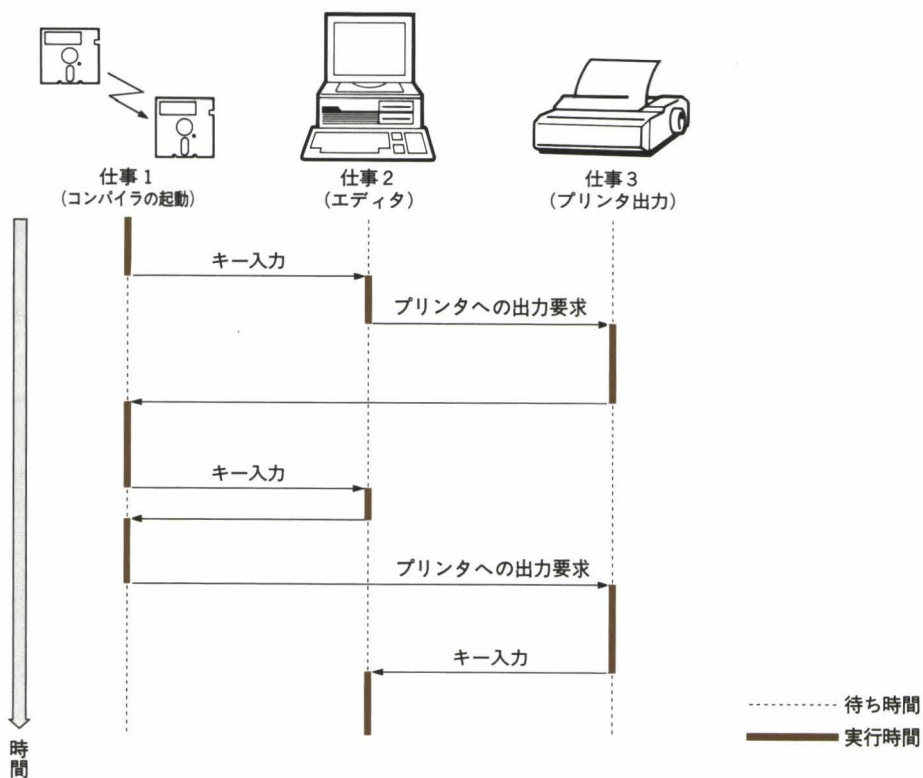



図6-1　時分割方式の概念

■ マルチタスク環境でのプログラミング

マルチタスク環境下ではどういったプログラムを組む必要があるのでしょうか？ 以下に，MS-DOSなどのシングルタスク・オペレーティングシステムでのプログラミングと比較した，マルチタスク環境下でのプログラミングについて注意点を挙げます．

①プリンタやディスクといったハードウェアはすべて共有資源なので，
- オペレーティングシステムの許可を受けてから使用しなければならない．
- アクセスを直接行ってはならない．すなわち，入出力はかならずシステムコールを利用する．

②メモリもシステム上で動いているプログラムすべての共有資源であるので，
- より少なく使う．
- 多く使う場合はその占有時間を小さくするよう努力する．
- 自分に許されたメモリ領域以外を使わない．

③タスクの数によりプログラムの実行速度が変化するため，
- ループ回数によって時間を計るなどのプログラムを記述しない．

①の例でいえば，第5章の割り込みを使った通信プログラムは，UNIX上では書いてはいけないことになります．マルチタスク環境では以上のような制限や利点に加えて，マルチタスク環境特有の次のような機能を理解しなければなりません．

- 1つのファイルを複数のタスクで書き込んだ場合はどうするのか(共有資源の排他機能)．
- 複数のプログラムどうしでのデータのやりとりはどうするのか(プロセス間通信機能)．
- 割り込み制御はどのように行われているのか．
- プロセス生成のメカニズムはどのようになっているのか．

■ UNIXのシステムコール

UNIXでは，前述の機能をすべてC言語から呼べるシステムコールとして実現しています．UNIXのシステムコールは以下のような種類に分類することができます．

- プロセス管理に関するシステムコール
- プロセス間通信に関するシステムコール
- メモリ管理に関するシステムコール
- 割り込みに関するシステムコール

次節ではこれらのうち，主要なシステムコールの概念とその使い方を取り上げます(とくに説明のある場合を除き，UNIX System V Release2/Release3の場合の解説をする)．

6.2　プロセス管理

■ プロセスとはなにか

UNIX における「プロセス」という言葉は，システムが現在動かしているプログラムのことをいいます．ファイルとしてディスク中にある実行形式のプログラムは「プロセス」とはいいません．さて，こういったプロセスはどのように実行されているのでしょうか？

UNIX には現在どのプロセスが動いているのかという情報を見るコマンドとして，ps というコマンドがあります．この ps コマンドを動かしてみましょう．

```
%ps -ef
     UID   PID  PPID  C   STIME TTY    TIME COMMAND
    root     0     0 18          ?  ? 171:04 swapper
    root     1     0  0          ?  ?   0:02 /etc/init
    root    38     1  0  Dec 13  co    0:01 - console m
    root    39     1  0  Dec 13  02    0:01 - tty02 m
    root    25     1  0  Dec 13   ?    0:01 /etc/update
      lp    30     1  0  Dec 13   ?    0:00 /usr/lib/lpsched
    root    34     1  0  Dec 13   ?    0:01 /etc/cron
    root    40     1  0  Dec 13  03    0:01 - tty03 m
    root    41     1  0  Dec 13  04    0:01 - tty04 m
    root    48     1  0  Dec 13  1a    0:01 - tty1a 6
    root   133   113 40 02:21:07 2A    0:02 ps -ef
    root   113     1  0 01:32:03 2A    0:03 -csh
%
```


図 6-2　ps コマンドの実行例

図 6-2 の最初の行の「swapper」は，メモリの仮想記憶機構をサポートするもので，システム全体のメモリを管理しているプロセスです．次の「init」はターミナルが tty に接続されているかどうかを調べて，「login：」のメッセージを出すプロセスを呼ぶプロセスです．

UNIX ではこのように各プロセスが順番をつけられて管理されています．この番号のことを「プロセス ID」と呼び，同じ 1 つの CPU 上では同一のプロセス ID を持ったプロセスは同時にはありません．また，プロセスはプロセス生成のシステムコールによって作られ，作られたプロセス自身の終了のシステムコールによって消滅します．以下に解説するシステムコールは，こういったプロセスの生成と消滅をつかさどるシステムコールです．

■ プロセス管理のシステムコール

プロセスを生成する2つのシステムコールを**表6-1**に示します．execXX関数には数種類ありますが，その違いは**表6-2**に示しています．

関数名	機　　能
fork( )	子プロセスを生成する
execXX( )	別の実行可能なプログラムをロードし実行する

表6-1　プロセスを生成するシステムコール

これらのシステムコールがプロセスからコールされると，コールされたプロセスから別のプロセスが作られます．execXX関数のみが実行されると，現在実行中のプロセスは消滅し，execXX関数で指定したプロセスが実行されます．

execXX関数の前にfork関数が実行されると，fork関数を実行したプロセスのコピーを作り，そのプロセスを起動します．つまり，fork関数が実行された直後には，まったく同じ2つのプロセスが同時に動き始めることになります．これらのプロセスをそれぞれ**親プロセス／子プロセス**と呼びます．

親プロセスと子プロセスで違う動作をするプログラムを作るには，たとえば**図6-3**のように記述します．forkに成功したら，子プロセス側のルーチンでexecXX関数を使えば，まったく別のプロセスを起動できることになります．

```
if(0 == fork())
    {
    子プロセスのプログラム
    }
else
    {
    親プロセスのプログラム
    }
```

図6-3　fork関数を使ったプログラム

execXX関数とfork関数の関係を次ページの**図6-4**に示します．起動したプロセスを終了するシステムコールには，表6-2に示す2つがあります．

関数名	機　　能
exit( )	プロセスが自分自身を終えるときに呼ぶ
wait( )	親プロセスが子プロセスの終了を待つ

表6-2　プロセスを終了するシステムコール

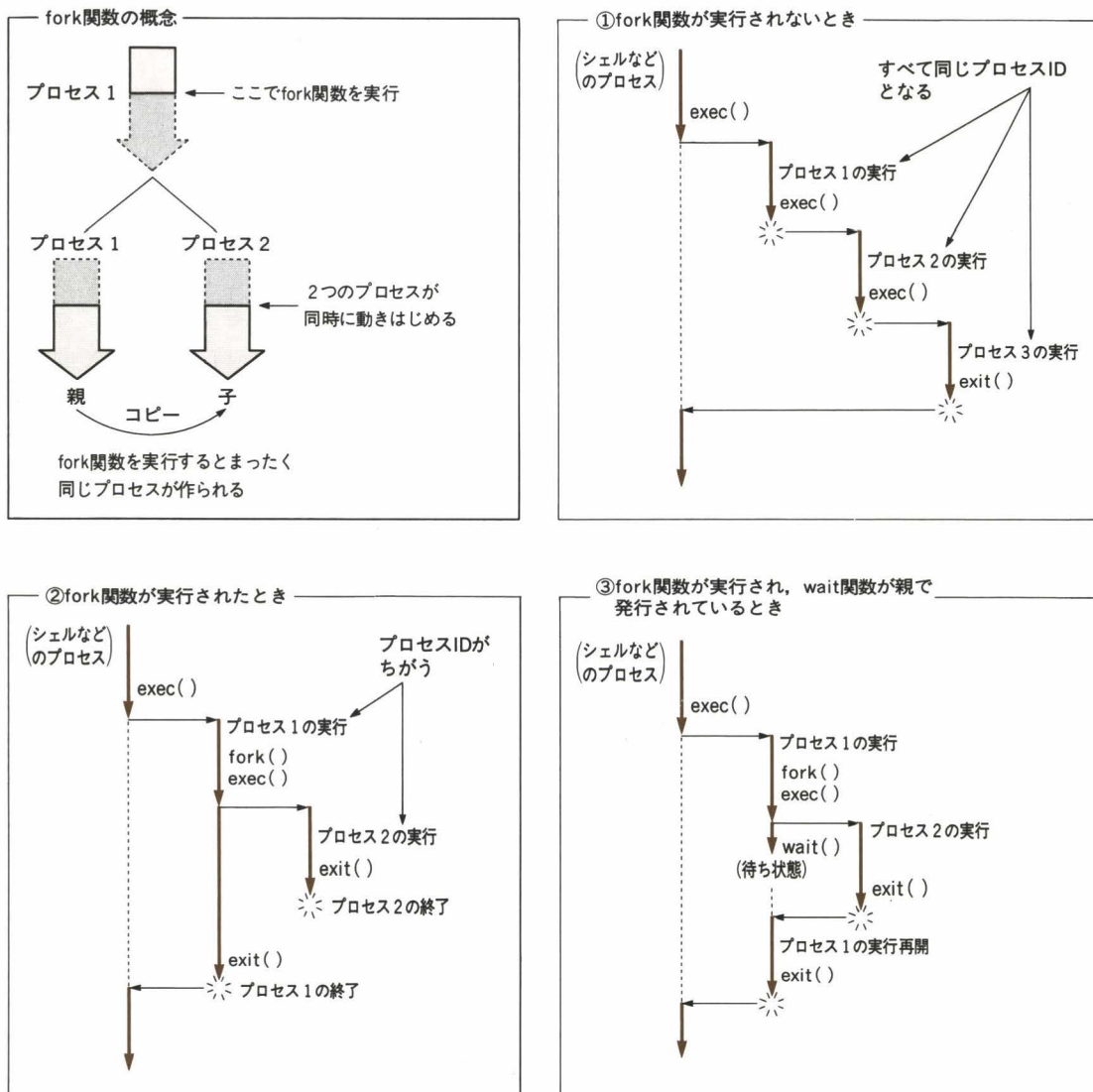



図6-4　プロセス生成システムコールの動作

表6-3にプロセス関係のシステムコールの書式や機能をまとめておきます．

関数名	書　式	返　値	機　能
fork( )	int fork( );	0=子プロセスである −1=親からのコール失敗 正の整数=子プロセスのプロセスID	子プロセスを生成する
execl( )	int execl(path, arg0, arg1, arg2, …, 0); char *path; ……実行ファイル名 char *arg0; char *arg1; プロセスに渡す引数 char *arg2; char *0; …………最後は0でくくっておく	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数を文字配列へのポインタとして渡す．これは起動されるプロセスのmain関数の引数として渡される
execv( )	int execv(path, argv); char *path; ………実行ファイル名 char **argv;……引数のポインタ配列へのポインタ．最後は0(NULLポインタ)	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数をポインタ配列へのポインタとして渡す
execle( )	int execle(path, arg0, arg1, …, 0, envp); char *path; ……….実行ファイル名 char *arg0; char *arg1; プロセスに渡す引数 char *arg2; char *0; ………………最後は0でくくっておく char **envp; ……環境変数のポインタ配列へのポインタ	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数を文字配列のポインタとして渡す．また，環境変数も渡す
execve( )	int execve(path, argv, envp); char *path; ……….実行ファイル名 char **argv; ……引数のポインタ配列へのポインタ．最後は0(NULLポインタ) cahr **envp; ……環境変数のポインタ配列へのポインタ．最後は0(NULLポインタ)	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数をポインタ配列へのポインタとして渡す
execlp( )	int execlp(path, arg0, arg1, arg2, …, 0); char *path; ……実行ファイル名 char *arg0; char *arg1; プロセスに渡す引数 char *arg2; char *0; …………最後は0でくくっておく	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数を文字配列のポインタとして渡す．コマンド実行時には，環境変数PATHをサーチする
execvp( )	int execvp(path, argv); char *path; ……….実行ファイル名 char **argv; ……引数のポインタ配列へのポインタ．最後は0(NULLポインタ)	−1=コール失敗	実行可能ファイルをロードし実行する．この際，引数をポインタ配列へのポインタとして渡す．コマンド実行時には，環境変数PATHをサーチする
exit( )	exit(stat); int stat;	なし	自プロセスを終了する．statには親プロセスに返すステータス値を入れる
wait( )	wait(stat); int *stat;	−1=コール失敗．あるいは自プロセスがシグナルを受けた 正の整数=子プロセスのプロセスID	子プロセスが終わるまで待つ．statには子プロセスのexit関数の引数として入れた値×0x100が返ってくる

表6-3 プロセス管理のシステムコール

プロセス管理のシステムコールを使ったサンプルプログラムを，**リスト 6-1** に紹介します．このプログラムでは，tr コマンドのように標準入力からのみ処理を行うコマンドを子プロセスとして起動し，指定されたファイルを標準入力へリダイレクトします．

```
 1: /*
 2:  * fork() & execXX() sample.
 3:  */
 4:
 5: #include <stdio.h>
 6:
 7: #define STDIN_FILDES    0
 8:
 9: int spawn_sample(progname, argument, stdin_name)
10: char *progname;
11: char **argument;
12: char *stdin_name;
13: {
14:     int fd;
15:     int pid, child;
16:     int status, reason;
17:
18:     if ((fd = open(stdin_name, 0)) < 0) { ……親プロセス側でファイルを
19:         perror(stdin_name);                  オープンしておく
20:         goto error;
21:     }
22:     if ((child = fork()) == 0) {
23:         /*
24:          * child process
25:          */
26:         close(STDIN_FILDES); ……標準入力をクローズし，先にオープンしたファイルを
27:         dup(fd);                割り当て，不要となったディスクリプタをクローズする
28:         close(fd);
29:         execvp(progname, argument); ……指定されたコマンドを実行する．成功した
30:         /*                              場合，この関数から戻ってこない
31:          * progname: no such file or not executable
32:          */
33:         perror(progname); ……実行不可能な場合には，エラーメッセージを表示する
34:         exit(1);
35:         /*NOTREACHED*/
36:     }
37:     /*
38:      * parent process
39:      */
40:     close(fd);      /* close child stdin */ ……指定されたファイルをクローズする
41:     if (child == -1) {
42:         /*
43:          * parent, but no more process ?   forkに失敗．カーネルのプロセス
44:          */                                 テーブルに空きがない
45:         perror("sorry");
46:         goto error;
47:     }
```

```
48:     do {
49:         if ((pid = wait(&status)) == -1) {  /* no children ? */
50:             perror("wait()");
51:             goto error;
52:         }
53:     } while (pid != child); ……………………目的の子プロセスの終了を待つ
54:     if ((status & 0377) != 0) { /* killed by some signal */
55:         if (status & 0200)   /* core dump */
56:             fprintf(stderr, "%d: core dumped¥n", child);
57:         goto error;
58:     }
59:     if ((reason = (status >> 8) & 0377) != 0)
60:         fprintf(stderr, "%d: exit %d¥n", child, reason); …子プロセスの終了
61:     return (reason);                                          ステータスの表示
62: error:
63:     return (-1);     /* failure */ ………………この関数内でエラーが起こった場合
64: }                                              には，ここに制御が移る
```

リスト 6-1　proc.c

```
 1: #include <stdio.h>
 2:
 3: main(c, v)
 4: char **v;
 5: {
 6:     int status;
 7:
 8:     if (c < 3) { ……………………………引数が不足している場合
 9:         fprintf(stderr, "usage: %s <file> <cmd> <args>¥n", v[0]);
10:         exit(1);
11:     }
12:     status = spawn_sample(v[2], &v[2], v[1]); ………v[c]はNULLであることを仮定
13:     fprintf(stderr, "%s: returned %d¥n", v[0], status);
14:     exit(0);
15: }
```

[実行結果]

```
% a.out main.c xyz a-z A-Z
xyz: No such file or directory
7655: exit 1 ……………………………子プロセスのプロセスIDと終了ステータスを表示
a.out: returned 1 ………………実行に失敗した場合１を返す
% a.out main.c tr a-z A-Z

#INCLUDE <STDIO.H>
    US = SPAWN_SAMPLE(V[2].
    FPRINTF(STDERR, "%S: RETURNED %D¥N", V[0].
    EXIT(0);
}
a.out: returned 0 ……………………………実行に成功した場合０を返す
%
```

リスト 6-2　main.c

6.3 セマフォ (System V系, 4.3BSD)

■ セマフォの概念

セマフォはどちらかというと，プロセス間通信のシステムコールですが，実際にはプロセス間の同期を取る目的で使われます．ここではプロセス制御として扱っていますが，簡単なプロセス間通信にも使えないことはありません．もちろん，後述するプロセス間通信のシステムコールがこのセマフォの代わりをしても問題はありません．

セマフォは以下の操作手順で使います．

①異なるプロセス間で同じキー番号を参照することによって，1つの共通なセマフォ値を持つ．
②キー番号で区別される個別セマフォは，ある既定値で初期化されている．このセマフォ値から1を引くという操作がセマフォへのアクセスとなる．
③セマフォを読みに行った結果，セマフォ値が負の値であればロックする．つまり，このオペレーションを行うシステムコールで待ちとなる．
④タスクの実行後，セマフォに1を加えるという操作を行う．

セマフォは，これ以外にもさまざまな操作が可能ですが，基本的な機能は，こういった操作によって，プロセスの実行を止めたり動かしたりすることです．図6-5にセマフォの概念図を示しておきます．

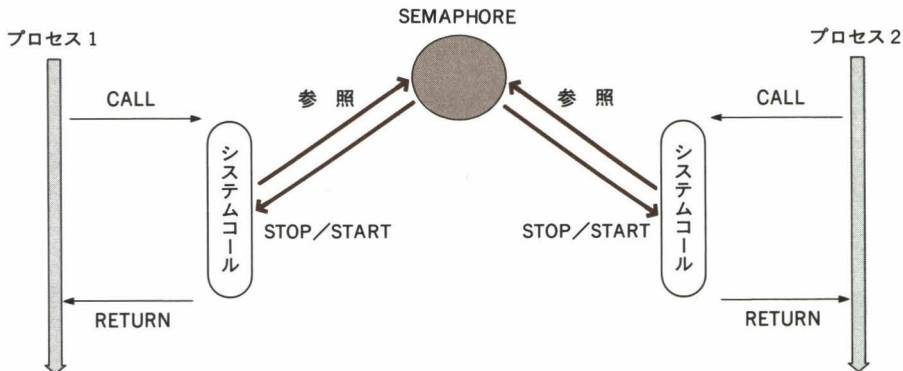



図6-5　セマフォの概念

■「入場制限型」セマフォの利用

セマフォは，たとえば1つのファイルを多くの並行して動くプロセス間で共有する場合などに，その**排他制御**として用いられます(図6-6).

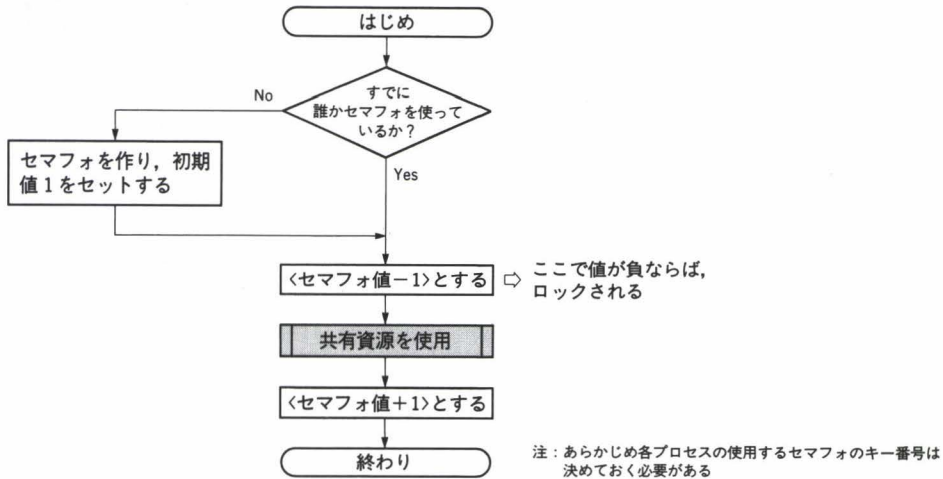



図6-6 セマフォによるファイルの排他制御

この制御でおもしろいのは，セマフォの初期値をもし1以外にしたらどうなるかということです．この1という数がもし2だったら，3だったらと考えると，セマフォの初期値はそのまま「リソースが共有できる最大のアクセス数」ということになります．つまり，ファイルなどは当然同時に1つの書き込みしか許されませんから1でよいわけですが，もし，ここに同時に2つの書き込みを許すリソースがあるとすると，そのときのセマフォの初期値は2にするのです．

このセマフォの初期値を利用すると，同時に動くある種のプロセス数の制限などにもセマフォが利用できることがわかるでしょう．たとえば，あるかなり重いプロセスがあるとします．このプロセスは，動作すると最初にセマフォ値を減らすようにすると，同時に動いている同じ種類のプロセス数に応じて，自らその実行を待つこともできるようになります．

■「交通信号機型」セマフォの利用

前述のようなセマフォの利用は，いわば「入場制限型の利用」とでも名づけることができます．これに対してプロセスの同期をとるための利用法もあります．たとえば，2つ以上のプロセスが，それぞれある瞬間に同期をとって動きたい場合などが当てはまります(図6-7).

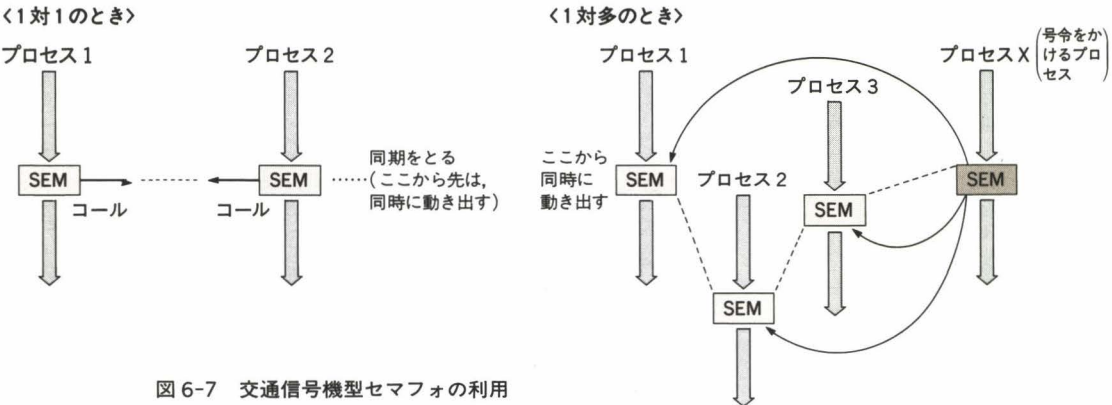



図6-7　交通信号機型セマフォの利用

セマフォのこういった使い方は，ほかのプロセス間通信機能でも実現が可能です．とくに1対多のときの利用などは，ほかのプロセス間通信機能を使う方が普通でしょう．

■ セマフォのシステムコール

表6-4にセマフォのシステムコールを示します．

<table>
<tr><th>関数名</th><th>書　式</th><th>返　値</th><th>機　能</th></tr>
<tr><td>semget( )</td><td>int semget(key, n, flg);
key_t key; ……オープンするセマフォのキー番号
int n; ……セマフォ数の指定
int flg; ……セマフォのフラグ</td><td>−1＝コール失敗
−1≠セマフォのID番号</td><td rowspan="2">セマフォのオープンを行う．この際，複数のセマフォを割り当てることができる(nで指定する)</td></tr>
<tr><td colspan="3">〈フラグの意味〉
IPC_ALLOC……すでに割り付けられているセマフォのみをアクセス
IPC_CREAT……すでに割り付けられたものでない場合，セマフォを作ってからアクセス
IPC_EXCL……すでに同じkeyのセマフォがある場合，エラーで返る
これらの値をセマフォのパーミッションビットとともにORをとってセットする
例：semget(key, n, 0666 | IPC_CREAT);</td></tr>
<tr><td>semop( )</td><td>int semop(id, oper, num);
int id; ……semget関数で得られたID番号
struct sembuf oper; ……セマフォ操作指示の構造体
int num; ……セマフォ操作数</td><td>−1＝コール失敗
0＝コール成功</td><td rowspan="2">セマフォの操作を行う．この際，複数のセマフォを一度に操作できる．セマフォ操作指示の構造体は左の形式で「/usr/include/sys/sem.h」に定義されている</td></tr>
<tr><td colspan="3">〈セマフォを操作する構造体の中身〉
struct sembuf {
unsigned short sem_num; ……実行するセマフォ番号(0～)
short sem_op; ……負の数で減，正の数で増，0でさらに同じセマフォ操作を繰り返す
short sem_flg; ……IPC_NOWAITを設定するとロックせずに，この操作からアプリケーションに戻ることができる
};</td></tr>
</table>

表6-4　セマフォのシステムコール(1)

この表で示したsemget関数，semop関数でセマフォ操作のほとんどは行えます．次の**表6-5**に示すsemctl関数は，もっと細かくセマフォを使いこなしたいときに使用するシステムコールです．

関数名	書式	返値
semctl( )	int semctl(id, num, com, array) ; int id ; ………semget関数で得られたID番号 int num ; ……セマフォ番号 int com ; ……操作コマンド array ; ………comによって指定された型．戻ってくるデータがはいる	－1＝コール失敗 0＝コール成功
	<comの型> GETVAL　現在のセマフォ値を得る SETVAL　セマフォ値を設定する GETPID　そのセマフォを操作した最後のプロセスIDを得る GETNCNT　現在セマフォ待ちをしているプロセス数を得る GETZCNT　現在セマフォ値が0になるのを待っているプロセス数を得る GETALL　seminfo／semid_ds構造体にセマフォのすべての状態を返す	

表6-5　セマフォのシステムコール(2)

ここで使われるセマフォを操作するための構造体の中身を，以下の図6-8と図6-9に示します．

```
 1: struct seminfo {
 2:         int     semmap; ……………………システム全体のセマフォ数
 3:         int     semmni; ……………………システム全体のセマフォID数
 4:         int     semmus; ……………………セマフォ数
 5:         int     semmnu; ……………………undo構造体の数
 6:         int     semmsl; ……………………ID当たりの最大セマフォ数
 7:         int     semopm; ……………………1回のsemop関数で操作可能な最大回数
 8:         int     semume; ……………………1プロセス当たりのundoの最大数
 9:         int     semusz; ……………………undo構造体のサイズ(バイト数)
10:         int     semvmx; ……………………最大のセマフォ数
11:         int     semaem; ……………………semadjの最大数
12: };
```

フラグ	機能
SETALL	seminfo構造体の値でセマフォの情報を設定する
IPC_STAT	セマフォIDで参照されるセマフォの情報をarrayで示すポインタの構造体に返してくる

図6-8　seminfo構造体の中身

```
1: struct semid_ds {
2:         struct   ipc_perm sem_perm;……………パーミッションの設定に関する構造体
3:         struct   sem      *sem_base;…………最初のセマフォへのポインタ
4:         unsigned short    sem_nsems;…………このIDでのセマフォ数
5:         time_t            sem_otime;…………最後にsemop関数を実行した時間
6:         time_t            sem_ctime;…………最後にセマフォ値を更新した時間
7: };
```

フラグ	機　　　能
IPC_SET	semid_ds構造体の情報を，指定したIDのセマフォに設定する
IPC_RMID	指定したIDのセマフォをシステムから消去する

図6-9　semid_ds構造体の中身

セマフォを使ったサンプルプログラムを**リスト6-3**と**リスト6-4**に示します．このプログラムでは，複数のプロセス間で共有できないコンソールというリソースをセマフォを使って管理する事例です．

```
 1: /*
 2:      Semaphore Using ……………………セマフォを使った画面出力プログラム
 3: */
 4:
 5: #include    <stdio.h>
 6: #include    <ctype.h>
 7: #include    <string.h>
 8: #include    <sys/types.h>
 9: #include    <sys/ipc.h>
10: #include    <sys/sem.h>
11:
12: #ifndef     BOOL
13: #define     BOOL        int
14: #define     FALSE       0
15: #define     TRUE        1
16: #endif      /* not defined BOOL */
17:
18: #define     S_KEY       ((key_t)12)……………セマフォのキー番号
19:
20:
21: main(argc,argv)
22: int     argc;
23: char    **argv;
24: {
25: int     i;
26: int     sem_id;
27: struct  sembuf   x_sem;
```

```
28:                                    セマフォのキーからID番号を得る
29: if((-1) == (sem_id = semget(S_KEY,1,IPC_EXCL | IPC_CREAT | 0666)))
30:     {
31:     if((-1) == (sem_id = semget(S_KEY,1,IPC_ALLOC | 0666)))
32:         {
33:         printf("¥7**** Cannot get semaphore-ID.¥n");
34:         exit(0);
35:         }                 すでに他のプロセスで，このセマフォを
36:     }                     使っていた場合は再度セマフォのID番号をとってくる
37: else    {                            このプロセスのセマフォがシステムで
38:     if((-1) == semctl(sem_id,0,SETVAL,1))……最初のものであったとき，セマフォ番
39:         {                            号0番に“1”の値をセットする
40:         printf("¥7**** Cannot set semaphore-value.¥n");
41:         exit(0);
42:         }
43:     }
44:
45: for(i = 0 ; i < 100 ; ++i)……メインループ
46:     {
47:     x_sem.sem_num = (unsigned short)0;……セマフォ番号の指定(0番)
48:     x_sem.sem_op  = (short)(-1);……セマフォのデクリメント・オペレーションの指定
49:     x_sem.sem_flg = (short)0;……IPC_NOWAITをセットしない
50:     if((-1) == semop(sem_id,&x_sem,1))
51:         {                                                     セマフォ
52:         printf("¥7**** Cannot Operation Semaphore for Dec.¥n");  のロック
53:         exit(0);
54:         }
55:                                                               排他制
56:     printf("pid = %05d : No.%03d : %s¥n",getpid(),i+1,argv[1]); 御され
57:                                                               る部分
58:     x_sem.sem_num = (unsigned short)0;
59:     x_sem.sem_op  = (short)1;……セマフォのインクリメント・オペレーションの指定
60:     x_sem.sem_flg = (short)0;……IPC_NOWAITをセットしない         セマフ
61:     if((-1) == semop(sem_id,&x_sem,1))                         ォのア
62:         {                                                     ンロッ
63:         printf("¥7**** Cannot Operation Semaphore for Inc.¥n");  ク
64:         exit(0);
65:         }
66:     }
67: }
```

リスト 6-3　sproc.c

```
 1: /*
 2:     Semaphore not Using……………………セマフォを使わない画面出力プログラム
 3: */
 4:
 5: #include     <stdio.h>
 6: #include     <ctype.h>
 7: #include     <string.h>
 8:
 9: #ifndef      BOOL
10: #define      BOOL        int
11: #define      FALSE       0
12: #define      TRUE        1
13: #endif       /* not defined BOOL */
14:
15:
16: main(argc,argv)
17: int     argc;
18: char    **argv;
19: {
20: int     i;
21:
22: for(i = 0 ; i < 100 ; ++i)
23:     {
24:     printf("pid = %05d : No.%03d : %s¥n",getpid(),i+1,argv[1]);
25:     }
26: }
```

リスト6-4　sproc2.c

```
% cc -O -s -Mel sproc.c -o sproc⏎   } オプションの－Melは，XENIX／286 System V
% cc -O -s -Mel sproc2.c -o sproc2⏎ } の場合のみ必要
%
% cat sem⏎ ……………………sproc.cを動かすシェルスクリプト
./sproc aaaa & ./sproc bbbbb & ./sproc cccccc & ¥
./sproc AAAA & ./sproc BBBBB & ./sproc CCCCCC & ¥
./sproc 1111 & ./sproc 22222 & ./sproc 333333
%
% cat sem2⏎ ……………………sproc2.cを動かすシェルスクリプト
./sproc2 aaaa & ./sproc2 bbbbb & ./sproc2 cccccc & ¥
./sproc2 AAAA & ./sproc2 BBBBB & ./sproc2 CCCCCC & ¥
./sproc2 1111 & ./sproc2 22222 & ./sproc2 333333
%
% sem2⏎ ……………………セマフォを使わない場合

pid = 02757 : No.018 : bbbbb
pid = 02757 : No.019 : bbbbb
pid = 02757 : No.020 : bpid = 02758 : No.001 : cccccc ……複数のプロセス間で
pid = 02758 : No.002 : cpid = 02759 : No.001 : AAAA      画面の取り合いが起
pid = 02759 : No.002 : AAAA                              こり，表示が乱れる
pid = 02760 : No.001 : BBBBB
pid = 02760 : No.002 : Bpid = 02761 : No.001 : CCCCCC
pid = 02761 : No.002 : Cpid = 02762 : No.001 : 1111
pid = 02762 : No.002 : 1pid = 02763 : No.001 : 22222pid = 02764 : No.001 : 33333
3
        :
pid = 02764 : No.002 : 3333pid = 02756 : No.005 : aaaa
pid = 02756 : No.006 : aaaa
pid = 02756 : No.007 : aaaa

pid = 02764 : No.099 : 333333
pid = 02764 : No.100 : 333333

%
% sem⏎ ……………………セマフォを使った場合

pid = 02766 : No.001 : aaaa
pid = 02766 : No.002 : aaaa
pid = 02767 : No.001 : bbbbb
        :
pid = 02767 : No.002 : bbbbb
pid = 02767 : No.003 : bbbbb
pid = 02767 : No.004 : bbbbb
pid = 02767 : No.005 : bbbbb ……………………複数のプロセスがセマフォにより同期をとって
pid = 02768 : No.004 : cccccc                 動いているので，画面が乱れない
pid = 02768 : No.005 : cccccc
pid = 02768 : No.006 : cccccc
pid = 02768 : No.007 : cccccc
        :
pid = 02768 : No.017 : cccccc
pid = 02768 : No.018 : cccccc
pid = 02768 : No.019 : cccccc

%
```

図 6-10 sproc.c と sproc2.c を起動するシェルスクリプトと実行結果

6.4 プロセス間通信

6.1節でも述べたように，マルチタスクのオペレーティングシステムでは，複数のプロセスを同時に実行できるだけでなく，各プロセス間で情報のやりとりを行い有機的に結合したアプリケーションプログラムを簡単に作成できることが大きな特徴です．ここではプロセス間通信の手法として，「パイプ」，「メッセージ」，「共有メモリ」の3つを取り上げ，その概念とシステムコールについて解説します．

■ パイプの概念

パイプは最も一般的に使われているプロセス間通信の方法です．これはシステムの共有エリア上にあるエリアを設け，FIFO(First-In First-Out)のデータ管理を行うようにしたものです．このエリアはファイルディスクリプタで管理されるため，通常のファイルのようにread関数やwrite関数によって読み書きを行うことができます．

パイプを使うにはその性質をよく把握しておくことが必要です．パイプは，forkされた子プロセスと親プロセスで同一のパイプバッファが使われるため，これを用いて親子関係のあるプロセス間で通信が行えます．また，親子関係のないプロセスどうしの通信には，デバイスとしてFIFOファイルを作り，それに対するアクセスを行うことによってプロセス間の通信を実現します．

パイプをデバイスファイルとする機能は「名前つきパイプ」(Named Pipe)と呼ばれ，次のコマンドで共通にアクセス可能なデバイスファイルを作成し，それを利用します．

```
%mknod  /dev/fifo  p
```

これでプロセス間通信を行うプロセスが「/dev/fifo」という名前のファイルにアクセスすることにより，FIFO形式でのシーケンシャルなデータの入出力を行うことができます(もちろん，ファイルのパーミッションの操作によって，ほかのプロセスがアクセスできないパイプも作れる)．パイプによる通信は，通常のファイルの入出力のコマンドでアクセスできるので，プログラム上も簡単になり有効なプロセス間通信の手段になります．

上述したパイプの概念を次ページの図6-11に示します．

注：パイプバッファは親子関係のあるプロセスグループに1つだけ与えられる．したがって，"このデータはどこから来てどこへ行くのか" というデータ管理は，パイプ自身ではいっさい行っていない．この制御はパイプではなく，パイプを呼ぶプロセス側で行わなければならない．

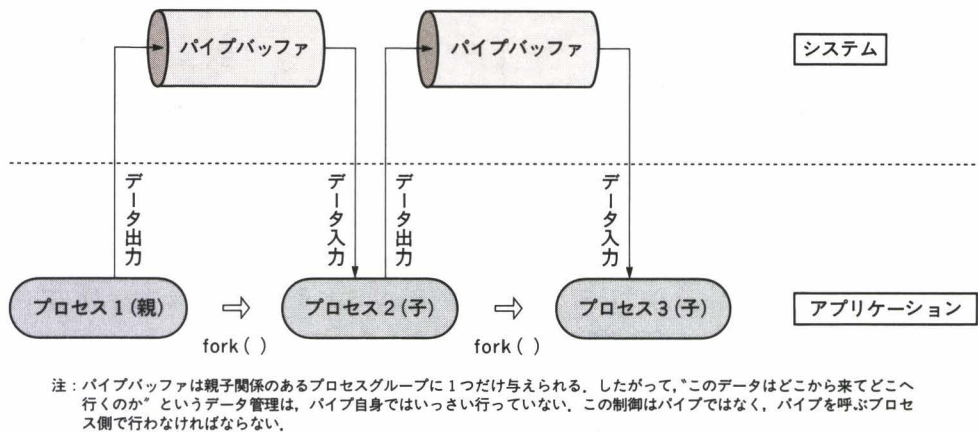



図 6-11　パイプの概念

■ パイプのシステムコール

パイプのシステムコールを**表 6-6** に示します．

関数名	書　式	返　値	機　能
pipe( )	int pipe (fildes) ; int fildes[2] ; fildes[0]＝入力用パイプのファイルディスクリプタ fildes[1]＝出力用パイプのファイルディスクリプタ	－1＝コール失敗 0＝コール成功	パイプ用のファイルディスクリプタを得る．ここで得られたディスクリプタをシステムに返すときはclose 関数を使う．またパイプへの書き込みはwrite関数を使い，パイプからの読み込みは read 関数を使う

表 6-6　パイプのシステムコール(1)

パイプの操作は，pipe 関数以外すべて低水準の入出力関数を使って行います．また，パイプバッファの量はシステムによって限られていますから，これらの制御もアプリケーション側で考慮しなければなりません．パイプがデータでいっぱいに詰まった状態や，からのパイプからデータを読み出した場合などの制御も必要でしょう．これらはすべて fnctl 関数(ファイルコントロール関数)などで O_NDELAY ビットを立てて制御するなどの方法を使って実現しなければなりません．

パイプのシステムコールには，pipe 関数以外に高水準のパイプ制御関数があります．それを次ページの**表 6-7** に示します．

関数名	書　式	返　値	機　能
popen( )	FILE ＊popen(com, type) ; char ＊com ;……起動コマンド名 char ＊type ;……リード/ライトのモード	0＝コール失敗 0≠コール成功 （ファイルポインタ）	通常のfopen関数と同じように設定するが，コマンドを起動し，そのコマンドとパイプ経由でのデータのやりとりを行う準備をする
pclose( )	int pclose(fp) ; FILE ＊fp ;………パイプのファイルポインタ	−1＝コール失敗 0＝コール成功	popen関数でオープンしたパイプをクローズする

表6-7　パイプのシステムコール(2)

パイプのシステムコールを使ったサンプルプログラムを**リスト6-5**に示します．このプログラムは，前述のリスト6-1に複数ファイルの指定機能をつけたものです．これはcatコマンドの出力結果を，パイプの機能を使って指定されたコマンドの標準入力へ受け渡す方法を用いています．

```
 1: #ifndef lint
 2: static char sccsid[] = "%W% %G%";   } SCCSによって展開されるキーワード
 3: #endif
 4:
 5: #include <stdio.h>
 6:
 7: #define CATENATE     "/bin/cat"
 8: #define CMD_SEP      "@"
 9: #define STDIN        0
10: #define STDOUT       1
11:
12: char    *progname;          /* command name at execution */
13:
14: main(argc, argv)
15: int argc;
16: char **argv;
17: {
18:     char *malloc();
19:     char **cmds;
20:     char **catv;
21:     int catc;
22:     int i;
23:     int status;
24:     int found;
25:
26:     progname = argv[0];
27:     if (argc < 3) {…………引数が不足している場合には使用法を表示
28:         fprintf(stderr,
29:             "usage: %s [<file>]... %s <cmd> [<args>]...¥n",
30:             progname, CMD_SEP);
31:         exit(1);
32:     }
```

```
33:     found = 0;                                          ┐
34:     for (catc = 1; catc < argc; catc++)                 │
35:         if (strcmp(argv[catc], CMD_SEP) == 0) {         │ ファイルとコマンドの
36:             found = 1;                                  │ 区切りを見つける
37:             break;                                      ┘
38:         }
39:     if (!found) { ……………コマンドの区切りがない
40:         fprintf(stderr, "%s: Missing command¥n", progname);
41:         exit(1);
42:     }
43:     cmds = &argv[catc+1];
44:     if ((catv = (char **)malloc(sizeof(char *)*(catc+1))) == NULL) {
45:         perror("sorry");            /* Not enough core */
46:         exit(1);
47:     }
48:     catv[0] = CATENATE;                  ┐
49:     for (i = 1; i < catc; i++)           │ catに渡す引数リストの作成
50:         catv[i] = argv[i];               │
51:     catv[catc] = NULL;                   ┘
52:     status = pipe_sample(catv, cmds);
53:     free((char *)catv);
54:     exit(status);
55:     /*NOTREACHED*/
56: }
57:
58: pipe_sample(catv, cmds)
59: char **catv;
60: char **cmds;
61: {
62:     int fds[2];             /* for pipe(2) call */
63:     int nchild;
64:     int cat_pid, cmd_pid, pid;
65:     int cat_status, cmd_status, status;
66:     int cat_reason, cmd_reason;
67:
68:     if (pipe(fds)) { ……………あらかじめ親プロセスでパイプを作っておく
69:         perror("pipe()");           /* File table overflow */
70:         return (-1);
71:     }
72:     if ((cat_pid = fork()) == 0) {                         ┐
73:         /* child process, 'cat' execute */                  │
74:         close(STDOUT);  ┐                                   │
75:         dup(fds[1]);    ┘標準出力のリダイレクト              │
76:         close(fds[0]);  ┐                                   │ catを実行
77:         close(fds[1]);  ┘不必要になったパイプをクローズ      │
78:         execv(catv[0], catv);                               │
79:         perror(catv[0]); ……………実行不可能なのでメッセージを表示 ┘
80:           exit(1);
81:         /*NOTREACHED*/
82:     }
```

```
 83:     if ((cmd_pid = fork()) == 0) {
 84:         /* child process, specified command */
 85:         close(STDIN);       } 標準入力のリダイレクト
 86:         dup(fds[0]);        }
 87:         close(fds[0]);      } 不必要になったパイプをクローズ
 88:         close(fds[1]);      }                                    ユーザーが指定した
 89:         execvp(cmds[0], cmds);                                   コマンドを実行
 90:         perror(cmds[0]);··················実行不可能なのでメッセージを表示
 91:          exit(1);
 92:         /*NOTREACHED*/
 93:     }
 94:     if (cat_pid == -1 || cmd_pid == -1) { ·············fork関数でエラーが発生した場合
 95:         perror("sorry");          /* No more processes */
 96:         goto error;
 97:     }
 98:     /* parent process */
 99:     close(fds[0]);  } 不必要となったパイプをクローズ，これは標準入力をリダイレクト
100:     close(fds[1]);  } されたプロセスにとってとくに重要
101:     /* wait for children */
102:     nchild = 2;
103:     do {
104: retry:
105:         if ((pid = wait(&status)) == -1) {
106:             perror("wait()");   /* No children */
107:             goto error;
108:         }
109:         if (pid == cat_pid)  ························catの子プロセスが終了
110:             cat_status = status;
111:         else if (pid == cmd_pid)··················ユーザーが指定した子プロセスが終了
112:             cmd_status = status;
113:         else
114:             goto retry;
115:     } while (--nchild > 0);
116:     cmd_reason = analize(cmd_pid, cmd_status);
117:     cat_reason = analize(cat_pid, cat_status);
118:     if (cmd_reason != 0)
119:         return (cmd_reason);
120:     if (cat_reason != 0)
121:         return (cat_reason);
122:     return (0);             /* success */
123: error:························なんらかのエラーが発生した場合にパイプをクローズしてから戻る
124:     close(fds[0]);
125:     close(fds[1]);
126:     return (-1);            /* something failure */
127: }
128:
129: analize(pid, status)··················子プロセスの終了ステータスの解析と表示
130: int pid, status;
131: {
132:     int term, reason;
133:
134:     if (status & 0200) {
135:         fprintf(stderr, "%d: core dumped¥n", pid);
136:         return (-1);
137:     }
```

```
138:      if ((term = status & 0377) != 0) {
139:          fprintf(stderr, "%d: killed by signal(%d)¥n", pid, term);
140:          return (-1);
141:      }
142:      if ((reason = (status >> 8) & 0377) != 0) {
143:          fprintf(stderr, "%d: exit %d¥n", pid, reason);
144:          return (reason);
145:      }
146:      return (0);       /* all right */
147: }
```

[実行結果]

```
% ls
main.c    multi.c
% cc multi.c -o multi
% ls
main.c    multi   multi.c
% multi
usage: multi [<file>]... @ <cmd> [<args>]...
% multi mian.c @ tr a-z A-Z  ……………………ファイル名を誤って指定した場合
mian.c: No such file or directory
8549: exit 1 ……………………子プロセスのプロセスIDと終了ステータスを表示
% multi main.c multi.c @ tr a-z A-Z  ……………………ファイルとコマンドは「@」で区切る

#INCLUDE <STDIO.H>

MAIN(C, V)
CHAR **V;
{
    INT STATUS;

    IF (C < 3) {
        FPRINTF(STDERR, "USAGE: %S <FILE> <CMD> <ARGS>¥N", V[0]);
        EXIT(1);
```

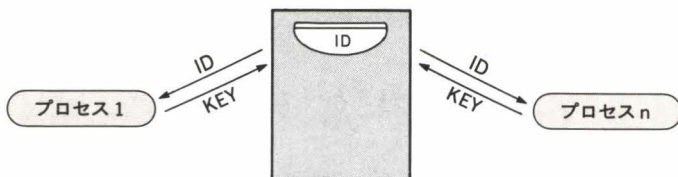
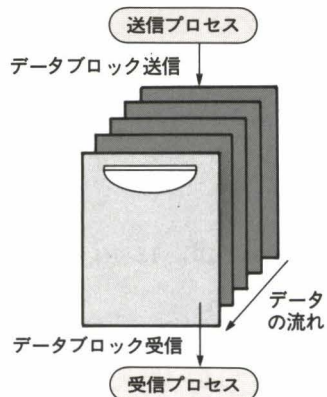
```
    INT TERM, REASON;

    IF (STATUS & 0200) {
        FPRINTF(STDERR, "%D: CORE DUMPED¥N", PID);
        RETURN(-1);
    }
    IF ((TERM = STATUS & 0377) != 0) {
        FPRINTF(STDERR, "%D: KILLED BY SIGNAL(%D)¥N", PID, TERM);
        RETURN(-1);
    }
    IF ((REASON = (STATUS >> 8) & 0377) != 0) {
        FPRINTF(STDERR, "%D: EXIT %D¥N", PID, REASON);
        EXIT(REASON);
    }
    RETURN(0);    /* ALL RIGHT */
}
%
```

リスト 6-5 multi.c

■ メッセージの概念(System V系, 4.3BSD)

メッセージをやりとりするシステムコールは，プロセス間通信の手段として提供されているもののうちでは高度なやりとりができるものの1つです．パイプは1バイトずつのデータのハンドリングしかせず，データ自身にIDをつけることもできないのに対して(もちろん，データそれ自身にそういうフィールドを持たせることはできるが)，このメッセージはまとまったデータにIDを持たせ，データをブロックで送受信することができます．また，親子関係のないプロセスどうしもつなげることが可能です．この機能によりプログラムの負荷が軽くなるばかりでなく，プログラム自身がすっきりしたものになる可能性があります．

図6-12にメッセージの概念図を示しておきます．

① メッセージIDはKEYによって取得する

② 以後メッセージへのアクセスはIDを使う

③ メッセージの送受信

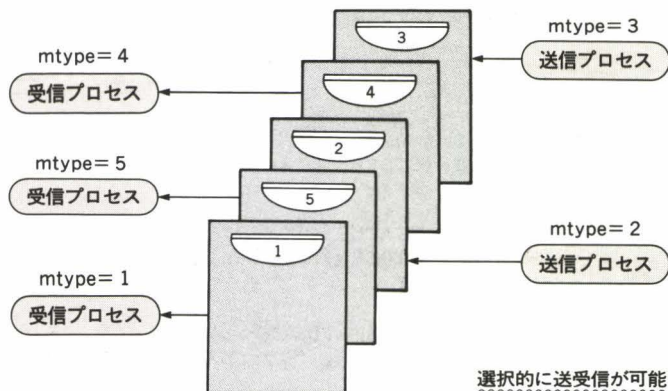



図6-12　メッセージの概念

■ メッセージのシステムコール

メッセージを使用可能にするシステムコールとメッセージの送信を行うシステムコールを**表 6-8**に示します．

関数名	書　式	返　値	機　能
msgget( )	int msgget(key, flg) ; key_t key ; ……メッセージのキー番号 int flg ; …………割り付け方法の指定とパーミッションの値のORをとったもの	－1＝コール失敗 －1≠コール成功 （メッセージIDが返る）	メッセージをプロセスが使えるように設定する．これによってメッセージIDを取得し，メッセージの送受信を行う．flgの値はセマフォの場合のフラグとまったく同じ（248 ページを参照）．同一プロセス内や親子関係のあるプロセス内でのみの使用には，さらにIPC_PRIVATEというフラグビットを立てておくと，他のプロセスからの侵入が起こらない
msgsnd( )	int msgsnd(id, dat, siz, flg) ; int id ; ……………………………メッセージのID番号 struct ……送信データの入った構造体のポインタ int siz ; ……………………………メッセージのバイトサイズ int flg ; ……………………………送信条件フラグ	－1＝コール失敗 0＝コール成功	メッセージの送信を行う

〈メッセージ本体のやりとりに使われる構造体〉

```
struct msgbuf {
        long   mtype ; …………テキストブロックにつくID番号
        char   mtext[n] ; ……テキスト本体
} ;
```

表 6-8　メッセージのシステムコール(1)

ここで大切なのは msgbuf 構造体の n という値，すなわちメッセージテキストの大きさです．多くの場合「/usr/include/sys/msg.h」に定義されていますが，とくに断わりのないかぎり，出荷時の設定は 1 になっています．つまりメッセージブロックの大きさは，自分で構造体を作って決めておいて，そのなかに定義して入れておかなければならないということです．この構造体にはメッセージテキストのポインタがメンバーになっているわけではなく，メッセージテキストを入れたバイト数のデータそのものが入るようにしなければなりません(つまり，この n という値を siz に入れておく)．また，flg に IPC_NOWAIT がセットされていると，メッセージバッファのあふれによるブロックを避けることができます．

表 6-9 にメッセージの受信を行うシステムコールとメッセージの制御を行うシステムコールを示します．

関数名	書　式	返　値	機　能
msgrcv( )	int msgrcv(id, dat, siz, typ, flg) ; int id ; ……………………メッセージのID番号 struct ı ……受信データの入った構造体のポインタ int siz ; ……………………メッセージのバイトサイズ int typ ; ……………………受信テキスト番号 int flg ; ……………………受信条件フラグ	－1＝コール失敗 －1≠コール成功 （受信バイト数）	メッセージを受信する．このとき，送信側で設定したmsgbuf構造体のmtypeと同じ番号のみを受信できる(typで指定する)．この値が0であれば，番号に関係なくデータを取り出す．flgの使い方はmsgsnd関数と同じで，ブロックされては困るプロセスで使う
msgctl( )	int msgctl(id, com, buf) ; int id ; ……………………メッセージのID番号 int com ; ……………………コマンド番号 struct msqid_ds ＊buf ; ……コントロール情報を入れる構造体のポインタ	－1＝コール失敗 －1≠コール成功 （コマンドによる返値）	メッセージの制御を行う．機能はcomによって決定

<comの型>

IPC_STAT ……………………メッセージに関する情報を得る
IPC_SET ……………………メッセージに関する情報をセットする
IPC_RMID ……………………メッセージを消去する
IPC_NOWAIT ……………………メッセージコールでブロッキングが起きないように指定する

<msgqid_ds構造体の内容>

```
struct msgqid_ds {
    struct ipc_perm msg_perm ; ………IPCのパーミッションに関する構造体
    unsigned short msg_qnum ; ………キューにつながっているメッセージ数
    unsigned short msg_qbytes ; ……キューにつながる最大バイト数
    unsigned short msg_lspid ; ………最後にメッセージを送ったプロセスID
    unsigned short msg_lrpid ; ………最後にメッセージを受けたプロセスID
    time_t msg_stime ; ……………………最後にメッセージを送った時間
    time_t msg_rtime ; ……………………最後にメッセージを受けた時間
    time_t msg_ctime ; ……………………最後にメッセージを更新した時間
} ;
```

表6-9　メッセージのシステムコール(2)

メッセージ関数を使ったサンプルプログラムは**リスト6-6～リスト6-12**のとおりです．このサンプルプログラムでは，1つのプロセスでユーザーに文字列を入力させ，それをもう1つのプロセスへメッセージ機能を使って送信し表示させます．2つのプロセスでの画面のコントロールには，エスケープシーケンスを用いています．

```
 1: #    makefile for Message call test procedure
 2: #    for XENIX/286 sysV
 3:
 4: OBJS0    =    mtest0.o mproc.o
 5: OBJS1    =    mtest1.o mproc.o
 6: CFLAGS = -O -s -W 1 -Mel……………………UNIX System Vでは，-Mel，-W1を削除
 7: LIBDIR = /usr/lib
 8:
 9: all:     mtest0 mtest1
10:
11: mtest0 :      $(OBJS0) mproc.h mymsg.h cur.h
12:          cc -Ml $(OBJS0) -o mtest0
13:                →リンクにあたってラージモデルのライブラリが必要なため指定する
14: mtest1 :      $(OBJS1) mproc.h mymsg.h cur.h
15:          cc -Ml $(OBJS1) -o mtest1

% mtest0 & mtest1⏎
```

リスト 6-6　Makefile

```
 1: /*
 2:     Message process - header……………………mproc.cを使うためのヘッダー
 3: */
 4:
 5: #ifndef      BOOL
 6: #define      BOOL         int
 7: #define      FALSE        0
 8: #define      TRUE         1
 9: #endif       /* not defined BOOL */
10:
11: #include     "mymsg.h"              /* My-msgbuf */
12:
13: BOOL     send();
14: BOOL     recv();
```

リスト 6-7　mproc.h

```
 1: /*
 2:     Message buffer definitions……………………mproc.cでメッセージのシステムコールを
 3:                                              使うためのヘッダー
 4:     Modified from SCO-XENIX286 sysV /usr/include/sys/msg.h
 5: */
 6:
 7: #define      MY_MSG_SIZE 256……………………送受信できる文字列のサイズ
 8:                           メッセージ・システムコールで使用する構造体．あらかじめ，
 9: struct   _my_msg {……………………このような器を用意しておく必要がある
10:     long     mtype; ………………メッセージのタイプ
11:     char     mtext[MY_MSG_SIZE];……送受信する文字列
12:     char     res;…………………………なくてもよいが，ワード／バイト・バウンダリの
13: };                                 調整のために入っている
14:
15: typedef struct   _my_msg      MY_MSG;……………typedefにより，あとで使用しやすくなる
```

リスト 6-8　mymsg.h

```
1: /*
2:      Easy-curses for all purpose……………………簡易CURSES用のマクロ
3: */
4:
5: #define c_save()     fputs("¥033[s",stdout);fflush(stdout)
6: #define c_rest()     fputs("¥033[u",stdout);fflush(stdout)
7: #define c_move(y,x) printf("¥033[%02d;%02dH",y+1,x+1);fflush(stdout)
8: #define c_cls()      fputs("¥n¥033[2J",stdout);fflush(stdout)
```

リスト6-9 cur.h

```
 1: /*
 2:      Message Call test procedure
 3: */
 4:
 5: #include      <stdio.h>
 6: #include      <string.h>
 7: #include      <ctype.h>
 8: #include      <sys/types.h>
 9: #include      <sys/ipc.h>
10: #include      <sys/msg.h>
11:
12: #include      "mymsg.h"              /* My-msgbuf */
13:
14: #define       MY_KEY        ((key_t)1)……………メッセージのキー番号は1
15: #define       MY_TYPE       ((long)2) ………………メッセージのタイプは2
16: #define       MY_FLAG       0 …………………………………メッセージのフラグはノーマル
17:
18: #ifndef       BOOL
19: #define       BOOL          int  ┐
20: #define       FALSE         0    ├ Bool型の定義
21: #define       TRUE          1    ┘
22: #endif        /* not defined BOOL */
23:
24:
25: /* ****************************************************************
26:      Send Messagees to Message-Que
27: **************************************************************** */
28:
29: BOOL     send(str)………………………メッセージの送信
30: char     *str;
31: {
32: int      msg_id;…………メッセージのID番号
33: MY_MSG   x_msg;………………送信するメッセージが長すぎる場合
34: int      i;
35:
36: if(MY_MSG_SIZE < strlen(str))…………………メッセージの送受信に使う構造体
37:      {
38:      return(FALSE);
39:      }
```

```
40:
41: for(i = 0 ; i < MY_MSG_SIZE ; ++i)………………メッセージバッファのクリア
42:     x_msg.mtext[i] = (char)0;
43:
44: strcpy(&(x_msg.mtext[0]),str);………構造体のメンバーへ送信する文字列をコピー
45: x_msg.mtype = MY_TYPE;…………………………メッセージのタイプをノーマルモードに設定
46:
47: if((-1) == (msg_id = msgget(MY_KEY,IPC_CREAT | 0666)))…キーからID番号を得る
48:     {
49:     return(FALSE);
50:     }
51:                                                          メッセージの送信…………
52: if(0 != msgsnd(msg_id,&x_msg,1 + strlen(&(x_msg.mtext[0])),MY_FLAG))
53:     {
54:     return(FALSE);
55:     }
56: else    return(TRUE);
57: }
58:
59:
60: /* ****************************************************************
61:     Get Messages from message-Que
62: **************************************************************** */
63:
64: BOOL    recv(str)…………………メッセージの受信
65: char    *str;
66: {
67: int     msg_id;
68: MY_MSG  x_msg;
69:
70: if((-1) == (msg_id = msgget(MY_KEY,IPC_CREAT | 0666)))…キーよりID番号を得る
71:     {
72:     return(FALSE);
73:     }
74:
75: if((-1) == msgrcv(msg_id,&x_msg,MY_MSG_SIZE,MY_TYPE,MY_FLAG))…………
76:     {
77:     return(FALSE);
78:     }
79:
80: strcpy(str,&(x_msg.mtext[0]));……………受信した文字列を文字配列strにコピーする
81:
82: return(TRUE);
83: }
```

リスト 6-10　mproc.c

```
 1: /*
 2:      Message-test procedure - SEND
 3: */
 4:
 5: #include    <stdio.h>
 6: #include    <ctype.h>
 7: #include    <string.h>
 8:
 9: #include    "./mproc.h"
10: #include    "./cur.h"
11:
12:
13: main() ……………………送信する文字列をキーボードから読み込み画面に表示したのち，実際に送信する
14: {
15: char     msg_buffer[MY_MSG_SIZE];
16: int      i;
17:
18: c_cls();…………………画面のクリア
19:
20: for(i = 0 ; i < 100 ; ++i)…………………メインルーチン
21:      {
22:      c_save();…………………カーソルの位置と属性のセーブ
23:      c_move(10,0);
24:      printf("%02d : SEND :
       ",i+1);
25:      c_rest();                                        文字列の入力
26:
27:      c_save();
28:      c_move(10,12);
29:      gets(msg_buffer);
30:      c_rest();
31:
32:      if(!send(msg_buffer))……………メッセージバッファへの送信
33:          continue;
34:
35:      if(0 == strlen(msg_buffer)) break;……………………送信した文字列の長さが0なら
36:      }                                                                   ばループを抜ける
37: c_save();
38: c_move(10,0);
39: printf("SEND : Send End. Send Null-Messages to another process.
            ");
40: c_rest();
41: c_move(24,0);
42: }
```

リスト6-11　mtest0.c

```
 1: /*
 2:      Message-test procedure - RECIEVE
 3: */
 4:
 5: #include     <stdio.h>
 6: #include     <ctype.h>
 7: #include     <string.h>
 8:
 9: #include     "./mproc.h"
10: #include     "./cur.h"
11:
12:
13: main()……………………………受信したメッセージを画面に表示
14: {
15: char    msg_buffer[MY_MSG_SIZE];
16: int     i;
17:
18: for(i = 0 ; i < 100 ; ++i)
19:      {
20:      if(!recv(msg_buffer))……………………メッセージの受信
21:           continue;
22:
23:      if(0 == strlen(msg_buffer))……………………メッセージの長さが0ならばループを抜ける
24:           break;
25:      else
26:           {
27:           c_save();                                 ┐
28:           c_move(12,0);                             │
29:           printf("                                  │
                    ");                                 │受信した文字
30:           c_move(12,0);                             │列の表示
31:           printf("%02d : RECV : %s",i+1,msg_buffer);│
32:           c_rest();                                 │
33:           }                                         ┘
34:      }
35:
36: c_save();
37: c_move(12,0);
38: printf("RECV : Null-Message received. Exit process.
               ");
39: c_rest();
40: c_move(24,0);
41: }
```

リスト 6-12　mtest1.c

■ 共有メモリの概念(System V系, 4.3BSD)

共有メモリ(Shared Memory)は，名前のとおり，複数のプロセス間で共有できるメモリエリアのことです．この共有メモリは構造が単純でわかりやすいだけでなく，さまざまなアプリケーション間の新しい通信手段を構築するツールの１つとして考えることができます．

また，直接カーネルのメモリを参照するためにアクセス速度が速く，スピードを要求されるアプリケーションどうしの通信にはかなり効果を発揮します．

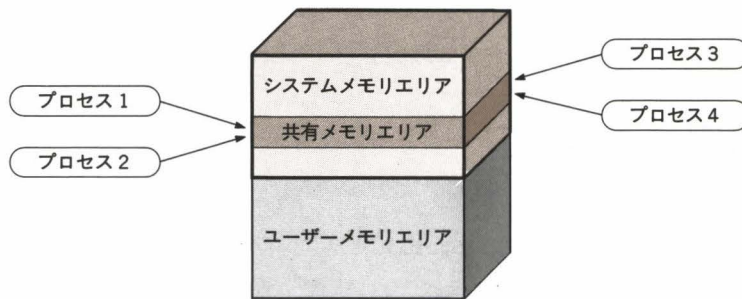



図 6-13　共有メモリの概念

共有メモリを使う場合は，前述のセマフォやメッセージのようにキーを指定して使うのですが，さらにもう１段階「アタッチ」という操作がはいり，そのあとで共有メモリエリアを使えるようになります．これは共有メモリが最終的にアプリケーションに受け渡すのが「ポインタ値」であるためです．つまり，

Key → ID → メモリエリアの先頭ポインタ

という順序でメモリエリアを得るわけです．システムでは，共有メモリも共有資源として管理していますから，こういった面倒な手続きが必要なわけです．

ところで，共有メモリや前述したメッセージ・システムコールのマクロ名に使われている IPC という言葉は，「Inter-Process Communication」(プロセス間通信)という術語を略したものです．

■ 共有メモリのシステムコール

表6-10に共有メモリのシステムコールを示します.

関数名	書　式	返　値	機　能
shmget( )	int shmget(key, siz, flg) ; key_t key ; ……キー番号 int siz ; …………エリアのデータサイズ int flg ; …………フラグ	−1＝コール失敗 −1≠コール成功 (共有メモリID)	共有メモリを要求する. flgの値はセマフォやメッセージのときとまったく同じ処理を行う
shmat( )	char ＊shmat(id, adr, opt) ; int id ; ……… shmget関数で得られたID番号 int adr ; ……エリアの先頭からのオフセット int opt ; ……オプション操作フラグ	−1＝コール失敗 −1≠コール成功 (共有メモリポインタ)	共有メモリのポインタを得る. optは読み出し専用とする場合にのみSHM_RDONLYの値をセットする. adrに0を入れると, 共有メモリの先頭の番地が得られる
shmdt( )	int shmdt(adr) ; int adr ; ……shmat関数で得られた共有メモリ番地	−1＝コール失敗 −1≠コール成功	shmat関数で受け取った共有メモリの使用をやめる
shmctl( )	int shmctl(id, com, buf) ; int id ; ……………………shmget関数で得られたID番号 int com ; ……………………………操作コマンド struct shmid_ds ＊buf ; ……共有メモリの情報を入れる構造体へのポインタ	−1＝コール失敗 0＝コール成功	共有メモリの制御をする. comの値によって以下の動作を行う

〈comの値〉

IPC_STAT ……………………………………………………共有メモリの管理情報を得る
IPC_SET ……………………………………………………共有メモリの管理情報をセットする
IPC_RMID ……………………………………………………共有メモリを消去する

〈shmid_ds構造体の中身〉

```
struct shmid_ds {
        struct ipc_perm shm_perm ; ……アクセス情報 (パーミッション)
        int shm_segs2 ; …………………………セグメントサイズ
        unsigned short shm_cpid ; ………共有メモリの作成プロセスのID
        unsigned short shm_lpid ; …………最後に操作をしたプロセスのID
        short shm_nattch ; ……………………現在アタッチ中のプロセス数
        time_t shm_atime ;  …………………最後にアタッチした時間
        time_t shm_dtime ; ……………………最後にデタッチした時間
        time_t shm_ctime ; ……………………最後に更新した時間
} ;
```

表6-10 共有メモリのシステムコール

6.5 メモリ管理

■ メモリ管理とUNIX

メモリ管理関数は，システム共有資源としてのメモリを有効に使うためのシステムコールです．このようなシステムコールを解放しているということは，ユーザーにシステム全体のパフォーマンスや社会的バランス感覚を考えに入れながらプログラムを組むことを要求していることを意味します．よく「UNIX はむずかしい」という声を聞きますが，UNIX の本当のむずかしさは，コマンドの名前がどうこういったレベルの問題ではなく，こういったメモリ管理に代表される〝システムのバランスをどうとるか〟といったむずかしさであるといってもよいでしょう．

■ malloc関数とその周辺関数

メモリ管理を行う関数は，実習 C 言語でも解説した malloc 関数，free 関数が主なものです．このほかにエリアの再割りつけを行う realloc 関数もあります．

これらの関数はマルチタスク特有のものではなく，その役割もシングルタスクのときとそれほど違いません．要するにオペレーティングシステムに必要なメモリ領域をもらったり返したりするわけです．しかし，マルチタスクでは多くのプロセスが同時に動いていますから，そのシステム中での〝社会的役割〟は違います．つまり，これらの使い方を誤ると，ほかのプロセスへの影響が大きいので，この章の冒頭で述べた原則を守って使うようにしなければなりません．

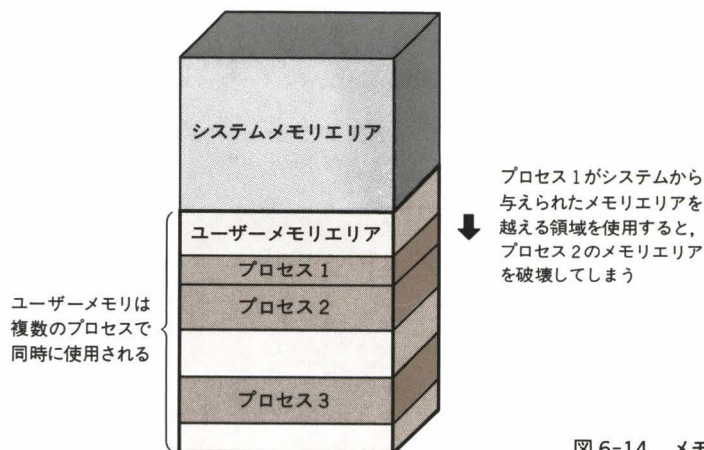



図 6-14　メモリ管理の概念

■ ブレークバリューとbrk関数

UNIX システムでの malloc 関数で得られるデータエリアは，システム規定値としての**ブレークバリュー**(Break Value)によって決められており，それ以上の大きさのエリアは割りつけることができません．このブレークバリューを知るには，ulimit 関数を使います．また，得られたブレークバリューをもとに，このブレークバリュー自身を変更することができます．つまり，1データセグメントで扱えるバイト数以上のエリアを使うことも可能になります．

表6-11 にメモリ管理を行うシステムコールの一覧を示します．

関数名	書　　式	返　　値	機　　能
malloc( )	char *malloc(size) ; unsigned int size ;	NULL＝割り当てに失敗 NULL≠割り当てたメモリエリアの先頭アドレスのポインタ	SIZEバイトのメモリ領域を割り当てる
free( )	void free(ptr) ; char *ptr ;	なし	malloc で割り当てたメモリを解放する
realloc( )	char *realloc(ptr, siz) ; char *ptr ; ………………エリアの先頭ポインタ unsigned int siz ; ……再割り付けするバイト数	0＝コール失敗 0≠コール成功 （メモリへのポインタ）	メモリエリアの再割りつけを行う
brk( )	int brk(end_ds) ; char *end_ds ; ……変更するデータセグメント値	－1＝コール失敗 0≠コール成功	end_dsの値は，新らたに設定されるデータセグメントの値だが，デフォルトでは現在の〈データセグメント値＋1〉になっている
sbrk( )	char *sbrk(inc) ; int inc ; ……データセグメント値の増分	－1＝コール失敗 －1≠コール成功 （古いセグメントの値）	このコールによって新しいセグメントが割りつけられる．brk関数で使うデータセグメント値のみを得たい場合はincの値を0とする
ulimit( )	int ulimit(com, new) ; int com ; ………コマンド番号 long new ; ……ファイルサイズの上限	－1＝コール失敗 －1≠コール成功	ファイル，メモリエリアのユーザーの限界値を得る
	〈comの値〉 com＝1 ……ファイルサイズの上限を返値として得る com＝2 ……ファイルサイズの上限値をnewの値とする com＝3 ……ブレークバリューの上限値を返値として得る		

表6-11　メモリ管理のシステムコール

6.6　割り込み処理

■ 割り込み処理の概念

マルチタスク・オペレーティングシステムでの割り込み処理は，すべてOSで管理されており，割り込みシグナルが直接ユーザーのアプリケーションプログラムに渡ることはありません．

外部や内部でシグナルが発生した場合，シグナルはまずオペレーティングシステムが受け取り，割り込みハンドリング・プロシージャがシステム中で統一された形式に整形して，アプリケーションに渡すことになっています．すなわち，すべてのアプリケーションに渡る割り込みは，ソフトウェア割り込みであるということです．この方法により，システム内外の割り込みをアプリケーションではすべて統一された形で扱えるわけです(図6-15)．

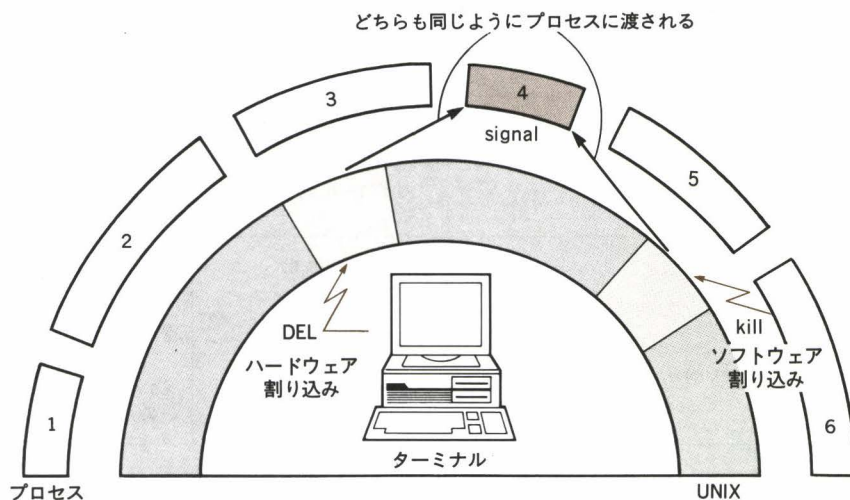



図6-15　割り込み処理の概念

プロセスがオペレーティングシステムからシグナルを受け取ると，基本的には以下の動作のいずれかを行います．

・プロセスを強制的に終了する．このときコアイメージ(デバッグ情報)をファイルに書き出す指定も可能．
・シグナルを無視する．
・あらかじめ指定しておいた(自分のプロセス内部の)関数に飛ぶ．

■ 割り込み処理のシステムコール

上記の動作を指定するのが**表 6-12** の signal 関数で，シグナルの種類はおよそ 20 種類ほどです．

関数名	書　式	返　値	機　能
signal( )	int （*signal(sig, func)）( )； int sig；……………………シグナル番号 int （*func）( )；……関数のアドレス	－1＝コール失敗 －1≠コール成功 （前の関数のアドレス）	シグナル処理の操作を行う

表 6-12　割り込み処理のシステムコール(1)

sig はシグナル番号ですが，以下のような値が「/usr/include/signal.h」に定義されています．

シグナル番号	機　能
SIGHUP	制御端末がなくなると発生する
SIGINT	設定された特殊なキー（通常はDELキー）がターミナルから押されると発生する
SIGQUIT	SIGINTと似ているが，コアダンプをともなう（通常はCTRL＋¥キー）
SIGILL	CPUの未定義命令の実行
SIGTRAP	デバッガのステップ，トレース動作のための割り込み
SIGIOT	abort関数を呼び出すと発生する
SIGEMT	浮動小数点演算用のプロセッサが存在しないときに発生する
SIGFPE	浮動小数点演算でのオーバーフロー，アンダーフロー，0での除算で発生する
SIGKILL	他のプロセスからkill（強制終了）されたときに発生する
SIGBUS	ハードウェアのトラブルで発生する
SIGSEGV	不正なメモリアクセスで発生する
SIGSYS	システムコールの引数を間違えると発生する
SIGPIPE	パイプのトラブルで発生する
SIGALRM	アラームクロックで発生する
SIGTERM	killコマンドを受けたときに発生する
SIGUSR1	ユーザーが独自に使えるシグナル
SIGUSR2	ユーザーが独自に使えるシグナル
SIGCLD	子プロセスが終了すると発生する
SIGPWR	外部電源が遮断されると発生する

表 6-13　シグナル番号の値

signal関数の2番目の引数funcは関数へのポインタですが，以下の値を入れると違う動きをします．ただし，これらの操作はできるシグナルとそうでないシグナルがありますから注意が必要です．

SIG_DFL ……割り込みを受けつける(そのまま終了する)
SIG_IGN ……割り込みを無視する

signal関数以外で割り込みに関係するシステムコールを**表6-14**に示します．

関数名	書　式	返　値	機　能
abort( )	void abort( );	なし	自分のプロセスにSIGIOTを送る．制御は戻ってこない．デバッグ時にcoreを生成させ，デバッガでトレースするために利用する
alarm( )	unsigned long alarm(sec); unsigned long sec; ……秒数	前回の残りのアラームカウント数	sec秒後にSIGALRMを発生する
pause( )	int pause( );	−1（どんな場合でも−1を返してくる）	シグナルを受けるまで待つ
kill( )	int kill(pid, sig); int pid; ……プロセスID int sig; ……シグナル番号	−1＝コール失敗 0＝コール成功	pidで指定したプロセスにsig番号のシグナルを送る．つまり，シグナルを人為的に発生させる．pidに0を指定すると自プロセスにシグナルがくる

表6-14　割り込み処理のシステムコール(2)

割り込みを使ったサンプルプログラムを**リスト6-13**に示します．このプログラムでは，指定したファイルを画面に表示し，CTRL+ZやDELなどの割り込み処理を行います．

```
 1: #ifndef lint
 2: static char sccsid[] = "%W% %G%";
 3: #endif
 4:
 5: #include <sys/types.h>
 6: #include <sys/ioctl.h>
 7: #include <sys/signal.h>
 8: #include <stdio.h>
 9:
10: #ifndef sigmask
11: #define sigmask(sig)        (1 << ((sig)-1))
12: #endif
13:
14: #define TTY_FD      2           /* stderr */
15: #define CMASK       0377        /* character mask */
16: #define MAXLINLEN   512
17: #define NLINES      22
```

```
18:
19: void    notty()
20: {
21:     perror("(stderr)");
22:     exit(1);
23:     /*NOTREACHED*/
24: }
25:
26: void    rawmode(on)
27: int     on;
28: {
29:     static  int raw = 0;…………………端末がrawモードになっていると1
30:     static  struct sgttyb save;……もともとの端末モード(退避用)
31:     struct  sgttyb buf;
32:
33:     if (on) {
34:         if (!raw) {
35:             if (ioctl(TTY_FD, TIOCGETP, &save))……………端末モードを得る
36:                 notty();
37:             buf = save;
38:             buf.sg_flags &= ~ECHO;
39:             buf.sg_flags |= CBREAK;…………1文字入力(入力した文字はエコーされない)
40:             raw = 1;
41:             if (ioctl(TTY_FD, TIOCSETN, &buf))……………端末モードの設定
42:                 notty();
43:         }
44:     } else {
45:         if (raw) {
46:             if (ioctl(TTY_FD, TIOCSETN, &save))…………端末モードの復帰
47:                 notty();
48:             raw = 0;
49:         }
50:     }
51: }
52:
53: /*ARGSUSED*/
54: int     onintr(sig)…………………DELキーなどのトラップ関数
55: int     sig;
56: {
57:     rawmode(0);…………端末モードをもとに戻す
58:     exit(1); ………………終了する
59:     /*NOTREACHED*/
60: }
61:
62: #ifdef SIGTSTP
63: int     ontstp(sig)…………………バークレイ版特有のジョブ・コントロール対応の関数
64: int     sig;
65: {
66:     int     mask;
67:
68:     rawmode(0);…………………………まず,端末モードを復帰
69:     signal(sig, SIG_DFL);………いったん停止するように変更
70:     mask = sigsetmask(sigblock(0) & ~sigmask(sig));………シグナルがブロック
71:     kill(0, sig);………自プロセスにstopシグナルを送り,いったん停止      されないように解除
72:     sigsetmask(mask); ……シグナルマスクをもとに戻す
73:     signal(sig, ontstp);‥トラップ関数を自関数にセット
74:     rawmode(1);……………………1文字入力モードへ
```

```
 75: }
 76: #endif
 77:
 78: void    setup()
 79: {
 80:     int     (*signal())();
 81:
 82:     if (signal(SIGHUP, SIG_IGN) != SIG_IGN)
 83:         signal(SIGHUP, onintr);
 84:     if (signal(SIGINT, SIG_IGN) != SIG_IGN)
 85:         signal(SIGINT, onintr);
 86:     if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)  B-Shellから起動された場合の対処
 87:         signal(SIGQUIT, onintr);
 88: #ifdef SIGTSTP
 89:     if (signal(SIGTSTP, SIG_IGN) != SIG_IGN)
 90:         signal(SIGTSTP, ontstp);
 91: #endif
 92:     rawmode(1);
 93: }
 94:
 95: void    done(status)
 96: int     status;
 97: {
 98:     rawmode(0);
 99:     exit(status);
100:     /*NOTREACHED*/
101: }
102:
103: int     readkey()……………………端末から1文字入力
104: {
105:     char    c;
106:
107:     if (read(TTY_FD, &c, 1) != 1) {
108:         abort();        /* don't happen */
109:         /*NOTREACHED*/
110:     }
111:     return (c & CMASK);
112: }
113:
114: void    view(fp)
115: FILE    *fp;
116: {
117:     char    buf[MAXLINLEN], *fgets();
118:     int     i;
119:
120:     do {
121:         for (i = 0; i < NLINES; i++) {
122:             if (fgets(buf, sizeof(buf), fp) == NULL)
123:                 goto brk;
124:             fputs(buf, stdout);
125:         }
126:         if (readkey() == 'q')
127:             break;
128:     } while (!feof(fp));
129: brk:;
130: }
```

```
131:
132: void    main(argc, argv)
133: int     argc;
134: char    **argv;
135: {
136:     FILE    *fp, *fopen();
137:
138:     setup();………………端末モードとシグナルの設定
139:     if (argc == 1)
140:         view(stdin);
141:     else
142:         while (--argc > 0) {
143:             if ((fp = fopen(*++argv, "r")) == NULL) {
144:                 perror(*argv);
145:                 continue;
146:             }
147:             view(fp);
148:             fclose(fp);
149:         }
150:     done(0);……………………端末モードをもとに戻して終了
151:     /*NOTREACHED*/
152: }
```

[実行結果]

```
% ls⏎
viewer.c
% cc viewer.c -o viewer⏎
% ls⏎
viewer      viewer.c
% viewer viewer.c⏎ ………………ソースファイルを画面へ表示
#ifndef lint
static char sccsid[] = "%W% %G%";
#endif

#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/signal.h>
#include <stdio.h>

#ifndef sigmask
#define sigmask(sig)        (1 << ((sig)-1))
#endif

#define TTY_FD      2           /* stderr */
#define CMASK       0377        /* character mask */
#define MAXLINLEN   512
#define NLINES      22

void    notty()
{
    perror("(stderr)");
    exit(1);
    /*NOTREACHED*/
}
```

```
void    rawmode(on)
int     on;
{
    static  int raw = 0;
^Z………………………CTRL+Zを押す

Stopped………………………いったん停止し，C-Shellに制御が戻る
% jobs
[1]  + Stopped                  viewer viewer.c
% fg
viewer viewer.c
    static  struct sgttyb save;
    struct  sgttyb buf;

    if (on) {
        if (!raw) {
            if (ioctl(TTY_FD, TIOCGETP, &save))
                notty();
            buf = save;
            buf.sg_flags &= ~ECHO;
            buf.sg_flags |= CBREAK;
            raw = 1;
            if (ioctl(TTY_FD, TIOCSETN, &buf))
                notty();
        }
    } else {
        if (raw) {
            if (ioctl(TTY_FD, TIOCSETN, &save))
                notty();
            raw = 0;
        }
    }
}

% jobs ………………………DELキーでコマンドを強制終了したのち，jobsコマンドを実行
%
```

リスト6-13　viewer.c

6.7　ソケット（4.3BSD）

4.3BSD 系の UNIX には，高度なプロセス間通信の方法として，ソケットと呼ばれる機能があります．その形式は「Named Pipe」と似ていますが，ソケットの場合はこれにメッセージの機能がついたものと思ってもらえればよいでしょう．

このソケットは共有メモリなどと比較して非常に扱いやすく，ネットワークで結合された複数の計算機間でも同様に動作します．ネットワーク機能を利用したアプリケーションの代表的なものとしては，rcp（ほかの計算機間でのファイルコピー），rlogin（ほかの計算機へのリモートログイン）などを挙げることができます．

■ クライアント／サーバーモデル

最近 X ウィンドウなどの説明でよく耳にする言葉に，「クライアント／サーバーモデル」（client/server model）があります．これは，クライアントというユーザー各自の環境を管理するプログラムと，システム全体の環境を保持しているサーバーというプログラムの 2 つに分けて処理を行うという概念です．UNIX において，クライアントは通常のプロセスですが，サーバーは daemon（デーモン）と呼ばれるプロセス形態を採用しています．このクライアント／サーバーモデルの利点は，kernel（カーネル）に変更を加えずに新しい機能を簡単に追加できるため，非常に柔軟性が高いことです．

■ socketコールと名前づけ

socket コールは，ファイルディスクリプタを 1 つ返し，このファイルディスクリプタに対して，読み書き（read/write）を行うことによって目的のプロセスと通信します（以後，ソケットといった場合はファイルディスクリプタを意味する）．UNIX の socket コールでは目的のプロセスを指定する方法として，プロセスに結合しているソケットに名前をつけるという方法が採用されています．この名前のつけ方は，各**ドメイン**（domain：名前空間）によって異なりますので注意してください．

ここで取り上げる Internet ドメインでソケットに名前をつけるには，まずすべての計算機にアドレス（Internet-address と呼ばれる形式）をつけ，さらに各計算機のソケットに対してポート番号（現在は 16 ビット）を割り振ります．さらに，Internet ドメインは，パケット単位の通信に向いている datagram（データグラム）と，バイトストリームで通信を行える virtual-circuit（バーチャルサーキット）に分けられています．

■ socketコールの利用

サーバーは通常システムのbootとともに起動され，ソケットを生成し予約された特定のポート番号を確保し，ソケットの監視を開始します．このポート番号は，各サービス(各アプリケーション)ごとに固有の番号が割り当てられています．

クライアントが起動されると，まずsocketコールによってソケットを生成します．そして，名前をつけるためにbindコールを呼び出し，自分のソケットに名前をつけます．サーバーでの処理が必要になった場合には，ソケットに対して通信を行い，処理を要求します．サーバーがこの要求を受け取って必要な処理を行い，結果をクライアントに送り返します．

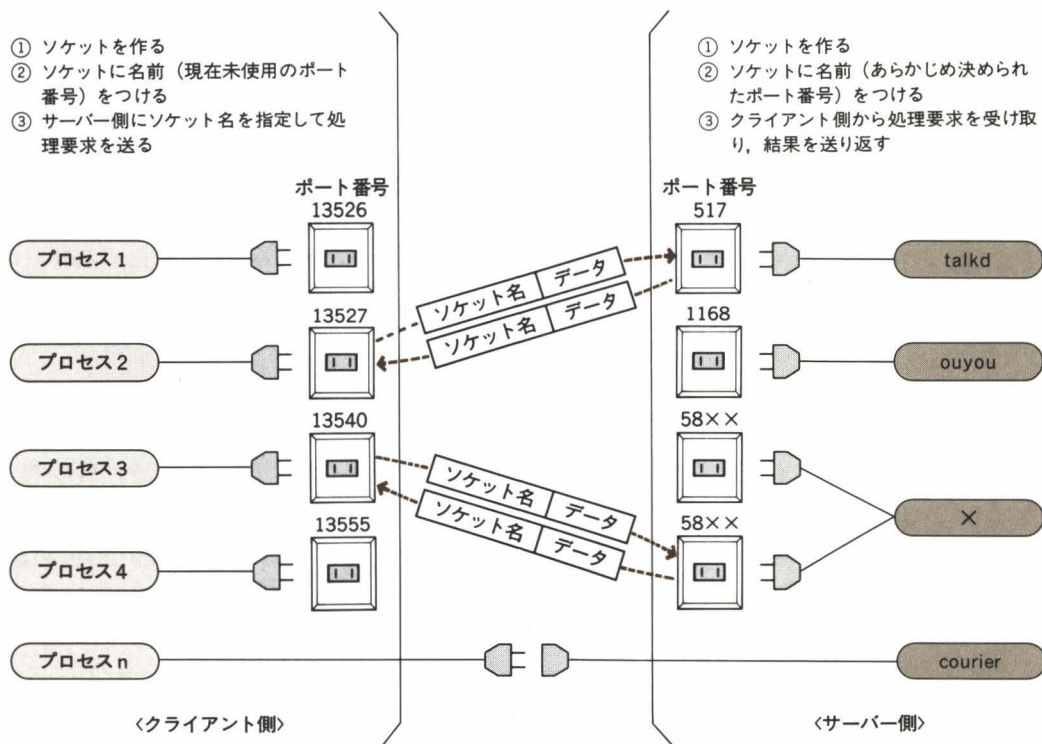



図6-16　ソケットを使ったプロセス間通信

■ ソケットのシステムコール

ソケットのシステムコールを表6-15に示します．これらの関数は，「sys/types.h」および，「sys/socket.h」をインクルードして使用します．

関数名	書　　式	返　値
select( )	int select(nfds, rdmask, wrmask, exmask, timeout) int nfds ; ……getdtablesize 関数で得た値を渡す int *rdmask, *wrmask, *exmask ; ……調べるファイルディスクリプタのビットマスク struct timeval *timeout ; ……一定時間待つ(NULLポインタで無限大)	有効なファイルディスクリプタの総数
	引数rdmask, wrmask, exmaskで指定したファイルディスクリプタに対して，入出力処理，および例外処理が可能かどうかを調べる	
socket( )	int socket(af, type, protocol) int af ; ……「AF_UNIX」，「AF_INET」などを指定 int type ; ……「SOCK_STREAM」，「SOCK_DREAM」，「SOCK_RAW」などを指定 int protocol ; ……通常は0を指定	ソケット (ファイルディスクリプタ)
	引数で指定されたソケットを生成	
bind( )	int bind(sock, name, length) int sock ; struct sockaddr *name ; ……<netinet/in.h> にあるsockaddr_in を指定† int length ; ……sizeof(struct sockaddr_in) を指定†	－1＝失敗 0＝成功
	socket関数で生成されたソケットに名前を付ける	
listen( )	int listen(sock, nqueue) int sock ; int nqueue ; ……ソケットへのコネクト要求をいくつまで待つか	－1＝失敗 0＝成功
	ソケットに対してのコネクト要求を最大いくつまで待つかを指定．最大値は <sys/socket.h> に，SOMAXCONN として定義されている(SOCK_STREAM)	
accept( )	int accept(sock, addr, lenptr) int sock ; struct sockaddr *addr ; ……結合した相手のソケット名 int *lenptr ; ……addr の大きさ	ソケット (ファイルディスクリプタ)
	相手プロセスからのコネクト要求を待ち，コネクト要求を受け取ると相手先のソケット名を格納し，新しいソケットを生成(SOCK_STREAM)	
connect( )	int connect(sock, name, length) int sock ; struct sockaddr *name ; ……結合する相手のソケット名 int length ; ……nameの大きさ	－1＝失敗 0＝成功
	sockによって指定されたソケットをnameで指定された相手プロセスのソケットに結合(SOCK_STREAM)	
recvfrom( )	int recvfrom(sock, buffer, buflen, flag, from, lenptr) int sock ; char *buffer ; } パケットを格納する領域 int buflen ; } int flag ; ……通常は0を指定 struct sockaddr *from ; } 発送先のアドレス† int *lenptr ; }	－1＝失敗 －1≠パケットの長さ
	ソケットに送られてきたパケットを引数buffer, buflenで指定された領域に格納し，送られてきた相手のアドレスを引数fromで指定された領域に格納(SOCK_DGRAM)	

関数名	書　　式	返　　値
sendto( )	int sendto(sock, buffer, buflen, flag, to, tolen) int sock ; char *buffer ; int buflen ; } パケットを格納する領域 int flag ; ……通常は0を指定 struct sockaddr *to ; int tolen ; } 送り先のアドレス†	−1＝失敗 −1≠パケットの長さ
	引数buffer, buflenで指定された領域に格納されたパケットを，引数toで指定された相手プロセスのソケットに送出(SOCK_DGRAM)	

† Internet ドメインの場合．UNIX ドメインでは異なるので注意

表6-15　ソケットのシステムコール

ソケット関連のシステムコールを利用するためには，このほかにいくつかのネットワーク関係の関数を知っておく必要があります．それらを表6-16に示します．

関数名	書　　式	返　　値
gethostname( )	int gethostname(hostname, length) char *hostname ; ……ホスト名 int length ; ……ホスト名の大きさ	−1＝失敗 0＝成功
	カレント計算機のホスト名を取得	
sethostname( )	int sethostname(hostname, length) char *hostname ; ……ホスト名 int length ; ……ホスト名の大きさ	−1＝失敗 0＝成功
	カレント計算機のホスト名を設定(rootのみが設定可)	
htonl( )	u_long htonl(hostlong) u_long hostlong ; ……longの場合	ネットワークオーダーでの値
	ホストオーダーからネットワークオーダーへの変換	
htons( )	u_short htons(hostshort) u_short hostshort ; ……shortの場合	ネットワークオーダーでの値
	ホストオーダーからネットワークオーダーへの変換	
ntohl( )	u_long ntohl(netlong) u_long netlong ; ……longの場合	ホストオーダーへの値
	ネットワークオーダーからホストオーダーへの変換	
ntohs( )	u_short ntohs(netshort) u_short netshort ; ……shortの場合	ホストオーダーへの値
	ネットワークオーダーからホストオーダーへの変換	
inet_ntoa( )	char *inet_ntoa(inet_style) struct in_addr inet_style ;	変換された文字列へのポインタ
	Internetアドレスからdot_notation(220.0.1.1のような文字列)への変更	
gethostbyname( )	struct hostent *gethostbyname(hostname)† char *hostname ;	構造体へのポインタ 0＝失敗
	ホスト名からInternetアドレスへの変換	

関数名	書　　　式	返　　値
getservbyname( )	struct servent *getservbyname(servname, protoname)[†] char *servname, *protoname ;	構造体へのポインタ 0=失敗
	サービス名からポート番号への変換	
getprotobyname( )	struct protoent *getprotobyname(protoname)[†] char *protoname ;	構造体へのポインタ 0=失敗
	プロトコル名からプロトコル番号への変換(socket関数の第3引数にこの結果を使う)	

† バイトオーダーの異なるメンバーもあるので注意

表 6-16　ネットワーク関係のシステムコール

■ サーバープログラムの解説

サーバープログラムは，以下の 3 つの条件コンパイルを選択できます．

① INETD

デーモンプロセスはシステムのブートとともに「/etc/rc」(または「/etc/rc.local」)から起動されるか，inetd デーモンから起動されるかのいずれかの形態をとります．"INETD"を定義してコンパイルすると inetd から起動できるようになります．「/etc/rc」から起動する場合には，inetd から起動される場合と比較して以下のような処理が必要です(これらについては後述)．

- 制御端末の切り離し
- サービスソケットの生成

なお，inetd から起動するためには，「/etc/inetd.conf」(4.3BSD)もしくは「/etc/servers」(Sun-OS)を変更する必要があります(図 6-17)．このファイルは root のみ変更が可能です．

<SERVICESでコンパイル>
/etc/servicesを追加 (4.3BSD, Sun-OS)

```
ouyou          1168/udp                        # test ouyou-C
```

<INETDでコンパイル>
/etc/inetd.confを追加 (4.3BSD)

```
ouyou   dgram   udp    wait    root    /usr/local/lib/ouyoud   ouyoud
```

/etc/servicを追加 (Sun-OS)

```
ouyou   udp     /usr/local/lib/ouyoud
```

図 6-17　「/etc/inetd.conf(/etc/servers)」の変更方法

② SYSLOG

ネットワークで結合された計算機環境の場合，デーモン内で生じるエラーメッセージはsyslog関数を使用して，適切な処置を施してくれる管理者に書き出す必要があります．inetdから起動した場合には，エラーメッセージを書きだせる標準エラーがないので，かならずSYSLOGを選択します．

③ SERVICE

ソケットを生成する際のポート番号を取得するために，「/etc/services」のデータベースを利用する場合に選択します．このデータベースもrootでなければ変更できません．なお，このデータベースを利用しない場合，ポート番号は定数としてサーバー／クライアントの両方のプログラムに埋め込むことになります．

前述の① inetdから起動しない場合には，以下のような前処理を行っておく必要があります．

setup_desc関数　…… 制御端末の切り離し

制御端末が接続されたままになっていると，キーボードシグナルを受ける可能性があるので，これを切り離します．しかし，いきなり制御端末を切り離すと自プロセスのプロセスグループ全体(「/etc/rc」を実行しているB-Shellも含む)がこの影響を受けますので，まずいったんfork関数によって子プロセスを生成し実際の処理は子プロセスで行い，親プロセスはすぐさま終了します．これによってサーバーの親プロセス(「/etc/rc」を実行しているB-Shell)が終了を検出し，先に処理を進めることができます．次に，自プロセスのプロセスグループをシステム内でユニークなもの(自プロセスのプロセスID)に変更します．以上の処理を行ったのちに，自プロセスの制御端末を切り離すことになります．

setup_sock関数　…… サービス用ソケットの生成

「/etc/services」のデータベースを利用する場合には，サービス名からポート番号を得るためにgetservbyname関数を呼び出します．このデータベースを利用せず，定数を埋め込む場合には，バイトオーダー変換用の関数htohsを使用してホストバイトオーダーからネットワークバイトオーダーに変換しなければなりません．また，ほかの計算機からの要求も受理できるように，計算機のアドレスを指定するsin_addr.s_addrにINADDR_ANYをセットしておきます．

次にsocket関数によってソケットを生成し，bind関数を呼び出して名前をつけます(既知のポート番号)．

inetdから起動する場合は，以上のような前処理を行う必要はありませんが，次に示す処理が必要です．

ready 関数 …… 要求到着の監視

通常サーバープロセスは，クライアントからの要求が到着するまでなにもすることがありません．この単なる待ち時間のあいだも，サーバープロセスが使用している主記憶は占有されたままとなります．このようなリソースのむだ使いをなくすために，一定時間(1分程度)クライアントからの要求が到着しない場合，そのままサーバープロセスを終了させます．これは，select 関数を用いて一定時間のタイムアウトを実現しています．これによって親プロセスである inetd がサーバーの代わりに I/O を待ち続けてくれます．

ready 関数では，一定時間内にクライアントからの要求が到着すれば，関数の返値として 1 を，到着しない場合には 0 を返します．また「/etc/rc」からの起動の場合は，常に 1 を関数の返値として返します．

これまで説明したきたサポート関数群によって，メインの service 関数は比較的すっきり記述できます．

service 関数 …… 要求を受け，処理した結果を送り返す

ready 関数を呼び出し，要求が到着しているかどうかを調べます．要求が到着していない場合には，この関数から戻ります．要求が到着している場合は，recvfrom 関数によって要求パケットと要求しているクライアントのソケット名(ポート番号と計算機のアドレス)を取得します．そしてこのパケットを処理関数である procedure 関数に引き渡し，実際の処理を行い，その結果をクライアントに対して送り返します．

main 関数 …… 環境の設定と service 関数の呼び出し

main 関数では，次の**表 6-17** のような条件によって処理を選択します．

	SYSLOG を選択	SYSLOG を選択しない
inetd を選択	①openlog関数を呼ぶ ②inetdから渡されるソケット(0)をグローバル変数serv_sockに代入	不可能
inetdを選択しない	①forkして制御端末を切り離す(setup_desc関数) ②openlog関数を呼ぶ ③setup_sock関数を呼ぶ	①setup_sock関数を呼ぶ

表 6-17　main 関数の条件

■ クライアントプログラムの解説

クライアントプログラムのコンパイルの条件は，SERVICE の 1 つだけです．これを定義してコンパイルすると，サーバープロセスと通信するための既知のポート番号を「/etc/services」のデータベー

スから得ます．これを選択しない場合，既知のポート番号は定数としてプログラムに埋め込みます．

setup 関数 …… サーバープロセスのアドレス決定とソケットの生成

サーバープロセスのソケットを指定するための変数 serv_addr を初期化します．サーバープログラムと同様にポート番号を得たあと(setup_sock 関数を参照)，指定された計算機のアドレスを取得するために gethostbyname 関数を呼び出します．そして serv_addr をゼロクリアしておき，さきほど取得した計算機のアドレスとポート番号を設定します．

ソケットを生成し名前(ポート番号)をつける場合は，サーバーの場合と異なり，既知のものである必要はありません．クライアントではソケットを一意に識別できる名前であればよいので，通常はポート番号のフィールドに 0(ネットワークバイトオーダー)を指定して bind 関数を使用し，現在未使用となっているポート番号を確保します．

transact 関数 …… サーバープロセスに要求を送り，その結果を受け取る

sendto 関数を用いてサーバープロセスに処理を要求します．この際サーバープロセスのアドレスを指定するために，setup 関数で初期化しておいたグローバル変数 serv_addr を指定します．そして recvfrom 関数を呼び出し結果を受け取ります．このとき，変数 pad にこの結果を送り返してきたプロセスのソケット名が格納されますが，今回のような場合には送り返してくるプロセスが明かなので，とくにチェックを必要としません．

```
 1: #CFLAGS = -DINETD -DSERVICE -DSYSLOG
 2: #CFLAGS = -DSERVICE -DSYSLOG
 3: #CFLAGS = -DSERVICE
 4: #CFLAGS = -DSYSLOG
 5: CFLAGS  =
 6:
 7: all:    program
 8:
 9: program: server client
10:
11: server: server.c
12:         $(CC) $(CFLAGS) server.c -o $@
13:
14: client: client.c
15:         $(CC) $(CFLAGS) client.c -o $@
16:
17: clean:; -rm -f server client
18:
19: # end of Makefile
```

リスト6-14 Makefile

```
 1: /* configuration check */
 2: #if defined(INETD) && !defined(SYSLOG)
 3:     not allowed.
 4: #endif
 5:
 6: #include <sys/types.h>
 7: #include <sys/socket.h>
 8: #include <netinet/in.h>
 9: #include <arpa/inet.h>
10:
11: #ifdef INETD
12: # include <sys/time.h>
13: #else
14: # include <sys/ioctl.h>
15: # include <sys/time.h>
16: # ifdef SERVICE
17: #  include <netdb.h>
18: # endif
19: #endif
20:
21: #ifdef SYSLOG
22: # include <syslog.h>
23: #else
24: # include <stdio.h>
25: #endif
26:
27: #ifndef NULL
28: # define NULL   0
29: #endif
30:
31: #ifdef INETD
32: # define TIMEOUT    60
33: #endif
34:
35: #ifdef SERVICE
36: # define SERV_NAME      "ouyou" ………サービス名はouyou
37: # define SERV_PROTO     "udp"          (「サーバプログラムの解説」の項を参照)
38: #else
39: # define SERV_PORT      1168 ………システムで一意の値に設定する
40: #endif
41:
42: #define MAXPACKETLEN    512
43: #define MAXHOSTLEN      256
44:
45: char    hostname[MAXHOSTLEN];
46: int     serv_sock = -1;
47:
48: #if !defined(INETD) && defined(SYSLOG)
49: void    setup_desc()
50: {
51:     int     fd;
52:                                   ………自プロセスのプロセスグループをユニークな
53:     setpgrp(0, getpid());               もの(ここではプロセスID)に変更する
54:     for (fd = getdtablesize()-1; fd >= 0; fd--) ……全ファイルディスクリプタ
55:         close(fd);                                   をクローズする
```

```
 56:     if ((fd = open("/dev/tty", 2)) >= 0) { ………自プロセスの制御端末をオープン
 57:         ioctl(fd, TIOCNOTTY, (char *)0); …………制御端末を切り離す
 58:         close(fd);
 59:     }
 60:     if ((fd = open("/dev/null", 2)) >= 0) {
 61:         if (fd != 0)
 62:             dup2(fd, 0);
 63:         if (fd != 1)
 64:             dup2(fd, 1);                        標準入出力には/dev/null
 65:         if (fd != 2)                            を割り当てておく
 66:             dup2(fd, 2);
 67:         if (fd != 0 && fd != 1 && fd != 2)
 68:             close(fd);
 69:     }
 70: }
 71: #endif
 72:
 73: void    done(status)
 74: int     status;
 75: {
 76: #ifdef notdef
 77:     if (serv_sock >= 0) {
 78:         shutdown(serv_sock, 2);
 79:         close(serv_sock);
 80:         serv_sock = -1;
 81:     }
 82: #endif
 83:       exit(status);
 84:     7*NOTREACHED*/
 85: }
 86:
 87: #ifndef SYSLOG
 88: char    *syserrmsg()
 89: {
 90:     extern  char *sys_errlist[];
 91:     extern  int sys_nerr;
 92:     extern  int errno;
 93:
 94:     if (errno < 0 || errno >= sys_nerr)
 95:         return ("unknown error");
 96:     return (sys_errlist[errno]);
 97: }
 98: #endif
 99:
100: #ifndef INETD
101: void    setup_sock()
102: {
103: # ifdef SERVICE
104:     struct  servent *sp, *getservbyname();
105: # endif
106:     struct  sockaddr_in sin;
107:
108: # ifdef SERVICE         ┌─→サービス名からポート番号を得る
109:     if ((sp = getservbyname(SERV_NAME, SERV_PROTO)) == NULL) {
```

```
110: #  ifdef SYSLOG
111:            syslog(LOG_ERR, "%s, %s: unknown service", SERV_NAME,
    SERV_PROTO);
112: #  else
113:            fprintf(stderr, "%s, %s: unknown service¥n", SERV_NAME,
    SERV_PROTO);
114: #  endif
115:            done(1);
116:            /*NOTREACHED*/
117:        }
118: # endif
119:        bzero((char *)&sin, sizeof(sin)); …………アドレス構造体をゼロクリアしておく
120:        sin.sin_family = AF_INET; ………………………アドレスフォーマットの指定
121: # ifdef SERVICE
122:        sin.sin_port = sp->s_port; ……………/etc/servicesから得たポート番号をセット
123: # else
124:        sin.sin_port = htons(SERV_PORT);………………固定のポート番号
125: # endif
126:        sin.sin_addr.s_addr = INADDR_ANY; …どの計算機からでも要求を受けられるようにする
127:        if ((serv_sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { …ソケットの
128: # ifdef SYSLOG                                                  生成(名前
129:            syslog(LOG_ERR, "socket: %m");                      なし)
130: # else
131:            perror("socket");
132: # endif
133:            done(1);
134:            /*NOTREACHED*/
135:        }                                       既知の名前をソケットにつける……
136:        if (bind(serv_sock, (struct sockaddr *)&sin, sizeof(sin))) { ……
137: # ifdef SYSLOG
138:            syslog(LOG_ERR, "bind; %d: %m", ntohs(sin.sin_port));
139: # else
140:            fprintf(stderr, "bind; %d: %s¥n", ntohs(sin.sin_port),
    syserrmsg());
141: # endif
142:            done(1);
143:            /*NOTREACHED*/
144:        }
145: }
146: #endif
147:
148: int     procedure(recvbuf, recvlen, sendbuf)
149: char    *recvbuf;
150: int     recvlen;
151: char    *sendbuf;
152: {
153:        strcpy(sendbuf, hostname);
154:        strcat(sendbuf, ":");                  送られてきたデータ(文字列)の直前に
155:        strncat(sendbuf, recvbuf, recvlen);    自計算機の名前を付加する
156:        return (strlen(sendbuf));
157: }
158:
```

```
159: int      ready()
160: {
161: #ifndef INETD
162:     return (1);     /* forever */ ………inetdデーモンを利用しない場合,
163: #else                                     無限ループする
164:     struct  timeval timeout;
165:     int readvec, nready;
166:
167:     timeout.tv_sec = TIMEOUT;………一定時間, 要求がない場合終了する
168:     timeout.tv_usec = 0L;
169:     readvec = 1 << serv_sock;
170:     if ((nready = select(serv_sock+1, &readvec, (int *)NULL,
    (int *)NULL, &timeout)) < 0) {
171: # ifdef SYSLOG
172:         syslog(LOG_ERR, "select: %m");
173: # else
174:         perror("select");
175: # endif
176:         return (0);
177:     }
178:     return (nready >= 1 && (readvec & (1 << serv_sock)));…要求があれば1
179: #endif                                                        なければ0
180: }
181:
182: void     service()
183: {
184:     char     *inet_ntoa( /* struct in_addr */ );
185:     struct   sockaddr_in from;
186:     int      fromlen;
187:     char     recvbuf[MAXPACKETLEN+1];
188:     int      recvlen;
189:     char     sendbuf[MAXPACKETLEN];
190:     int      sendlen;
191:
192:     while (ready()) {
193:         fromlen = sizeof(from);→クライアントプロセスからの要求を読み込む
194:         if ((recvlen = recvfrom(serv_sock, recvbuf, MAXPACKETLEN,
    0, (struct sockaddr *)&from, &fromlen)) < 0) {
195: #ifdef SYSLOG
196:             syslog(LOG_ERR, "recvfrom: %m");
197: #else
198:             perror("recvfrom");
199: #endif
200:             continue;
201:         }
202:         sendlen = procedure(recvbuf, recvlen, sendbuf);………要求を処理する
203:         if (sendto(serv_sock, sendbuf, sendlen, 0, (struct
    sockaddr *)&from, fromlen) != sendlen) {………結果を送り返す
204: #ifdef SYSLOG
205:             syslog(LOG_ERR, "sendto; %s: %m", inet_ntoa(
    from.sin_addr));
206: #else
207:             fprintf(stderr, "sendto; %s: %s¥n", inet_ntoa(
    from.sin_addr), syserrmsg());
208: #endif
209:             continue;
```

```
210:             }
211:         }
212: }
213:
214: void     main(argc, argv)
215: int      argc;
216: char     **argv;
217: {
218: #if !defined(INETD) && defined(SYSLOG)
219:     int pid;
220:
221:     if ((pid = fork()) != 0) {
222:         /* parent process */
223:         if (pid == -1) {
224:             perror("fork");
225:             exit(1);
226:             /*NOTREACHED*/
227:         }
228:         exit(0);
229:         /*NOTREACHED*/
230:     }
231:     setup_desc();……………………プロセスグループの変更，制御端末の切り離し
232: #endif
233: #ifdef SYSLOG
234:     openlog(argv[0], LOG_PID);………………syslog関数の使用宣言
235: #endif
236: #ifndef INETD
237:     setup_sock();……………………自力でサービスソケットを生成する
238: #else
239:     serv_sock = 0;        /* from 'INETD(8C)' */
240: #endif
241:     gethostname(hostname, sizeof(hostname));……自計算機の名前をセットしておく
242:     service();……………………サービスを開始する
243: #ifdef SYSLOG
244:     closelog();……………………syslog関数の終了宣言
245: #endif
246:     done(0);
247: }
```

(221〜230行) 制御端末を切り離す場合，親プロセスも影響を受けるので，子プロセスを生成し，小プロセス側で実行の処理を行う．親プロセスは即座に終了する

リスト6-15 server.c

```
 1: #include <sys/types.h>
 2: #include <sys/socket.h>
 3: #include <netinet/in.h>
 4: #include <stdio.h>
 5: #include <netdb.h>
 6:
 7: #ifdef SERVICE
 8: # define SERV_NAME   "ouyou"…………サービス名はouyou(「サーバプログラムの解説」の項を参照)
 9: # define SERV_PROTO  "udp"
10: #else
11: # define SERV_PORT   1168………………システムで一意の値に設定する
12: #endif
13:
14: #define INPORT_ANY   0
15: #define MAXHOSTLEN   256
16: #define MAXLINLEN    512
17:
18: struct  sockaddr_in serv_addr;
19: int     serv_sock = -1;
20:
21: void    done(status)
22: int     status;
23: {
24: #ifdef notdef
25:     if (serv_sock >= 0) {
26:         shutdown(serv_sock, 2);
27:         close(serv_sock);
28:         serv_sock = -1;
29:     }
30: #endif
31:     exit(status);
32:     /*NOTREACHED*/
33: }
34:
35: void    setup(servhost)
36: char    *servhost;
37: {
38: #ifdef SERVICE
39:     struct  servent *sp, *getservbyname();………サービス名からポート番号
40: #endif                                          を得る関数
41:     struct  hostent *hp, *gethostbyname();………ホスト名からInternetアドレスを
42:     struct  sockaddr_in sin;                    得る関数
43:
44: #ifdef SERVICE
45:     if ((sp = getservbyname(SERV_NAME, SERV_PROTO)) == NULL) {
46:         fprintf(stderr, "%s, %s: unknown service¥n", SERV_NAME,
    SERV_PROTO);
47:         done(1);
48:     }
49: #endif
50:     if ((hp = gethostbyname(servhost)) == NULL) {
51:         fprintf(stderr, "%s: unknown host¥n", servhost);
52:         done(1);
53:     }
```

```
 54:     bzero((char *)&serv_addr, sizeof(serv_addr));…アドレス構造体をゼロクリア
                                                          しておく
 55:     serv_addr.sin_family = hp->h_addrtype;……アドレス・フォーマットを指定
 56: #ifdef SERVICE
 57:     serv_addr.sin_port = sp->s_port;……データベースから得たポート番号
 58: #else
 59:     serv_addr.sin_port = htons(SERV_PORT);……固定のポート番号
 60: #endif
 61:     bcopy(hp->h_addr, (char *)&serv_addr.sin_addr, hp->h_length);…
 62:     bzero((char *)&sin, sizeof(sin));           接続先の計算機のアドレスをセット…
 63:     sin.sin_family = hp->h_addrtype;
 64:     sin.sin_port = INPORT_ANY;……現在，未使用のポート番号の確保
 65:     sin.sin_addr.s_addr = INADDR_ANY;
 66:     if ((serv_sock = socket(serv_addr.sin_family, SOCK_DGRAM, 0))
   < 0) {                  └→ソケットの生成(名前なし)
 67:         perror("socket");
 68:         done(1);
 69:     }
 70:     if (bind(serv_sock, (struct sockaddr *)&sin, sizeof(sin))) {……
 71:         perror("bind");
 72:         done(1);                        名前なしソケットに名前をつける……
 73:     }
 74: }
 75:
 76: void    transact(sendbuf, recvbuf)
 77: char    *sendbuf;
 78: char    recvbuf[MAXLINLEN];
 79: {
 80:     struct  sockaddr_in pad;
 81:     int     padlen;
 82:     int     length;
 83:
 84:     length = strlen(sendbuf) + 1;
 85:     if (sendto(serv_sock, sendbuf, length, 0, (struct
   sockaddr *)&serv_addr, sizeof(serv_addr)) != length) {……要求をサーバー
 86:         perror("sendto");                                  プロセスに送る
 87:         done(1);
 88:     }
 89:     padlen = sizeof(pad);
 90:     if ((length = recvfrom(serv_sock, recvbuf, MAXLINLEN-1, 0,
   (struct sockaddr *)&pad, &padlen)) < 0) {……結果をサーバープロセスから得る
 91:         perror("recvfrom");
 92:         done(1);
 93:     }
 94:     recvbuf[length] = '¥0';
 95: }
 96:
 97: void    filecopy(fp)
 98: FILE    *fp;
 99: {
100:     char    sendbuf[MAXLINLEN], recvbuf[MAXLINLEN], *fgets();
101:
102:     while (fgets(sendbuf, sizeof(sendbuf), fp) != NULL) {……1行ファイル
103:         transact(sendbuf, recvbuf);……サーバープロセスで処理させる  から読み込む
104:         fputs(recvbuf, stdout);……結果を標準出力に書き出す
105:     }
106: }
```

```
107:
108: void    main(argc, argv)
109: int     argc;
110: char    **argv;
111: {
112:     FILE    *fp, *fopen();
113:     char    hostname[MAXHOSTLEN];
114:
115:     if (argc >= 2 && strcmp(argv[1], "-h") == 0) {
116:         if (argc < 3) {
117:             fprintf(stderr, "missing hostname¥n");
118:             exit(1);
119:         }
120:         setup(argv[2]);
121:         argc -= 2;
122:         argv += 2;
123:     } else {
124:         gethostname(hostname, sizeof(hostname));
125:         setup(hostname);
126:     }
127:     if (argc == 1)
128:         filecopy(stdin);
129:     else
130:         while (--argc > 0) {
131:             if ((fp = fopen(*++argv, "r")) == NULL) {
132:                 perror(*argv);
133:                 continue;
134:             }
135:             filecopy(fp);
136:             fclose(fp);
137:         }
138:     done(0);
139: }
```

115: 接続計算機を指定した場合
120: 指定された計算機と接続できるように変数を設定
124–125: とくに指定されない場合，ローカルな計算機と接続

リスト 6-16　client.c

第7章
プログラム開発環境

この章では，プログラム開発環境について解説します．
大型コンピュータが導入されはじめた時代から，8ビットCPUのパソコンが主流であった数年前まで，「開発に使用するプログラム」というとコンパイラとリンカ，アセンブラ，エディタ程度のものしかないのが普通でした．しかし最近ではこういった状況も改善されてきて，いろいろな開発ツールが用意されるようになっています．

プログラムを開発するのに必要な〝プログラム〟とその整備は，「環境」ということばで表されるとおり，その1つ1つをとってみてもなんの意味もありません．これまで，日々の仕事のなかでプログラマが必要であると思ったときに必要な道具を自分で開発し，仕事をより確実に，そしてより能率を上げるという行為は非生産的であるように思われてきました．しかし，生産物が不可視でかつ複雑なソフトウェアの世界では，その規模が膨大になればなるほどこういった隠れた開発物が大きな役割を果たしてきます．

この章では，一般的になりつつあるソフトウェアツールのなかから，MAKEとSYMDEB，UNIX用のadbについて解説します．

7.1 MAKE — コンパイル／リンク自動化ツール —

MAKEは分割コンパイルのためのツールです．このコマンドのもとは，多くのソフトウェアツールがそうであるように，やはりUNIXにあります．MAKEは，便利なツールであることと，構造が比較的簡単で，そのソースファイル自身もC言語で記述されており移植性が高いことから，数々のUNIX以外のオペレーティングシステムにも移植されたり，真似をされたりしてきました．

ここでは，MS-DOSのMAKE(PC-9801シリーズのMS-DOS Ver3.1以降に付属のもの，またはマクロアセンブラやMS-CなどのCコンパイラに付属のもの)と，数々のバージョンのMAKEのもとなったUNIXのmakeについて解説します．

1990年10月現在，MS-DOS用のmakeはかなりの種類のものが出回っています．
特にCコンパイラに付属のmakeなども最近は特に多くなっており，そのすべてをここでご紹介することは困難です．しかし，これらのmakeは大きく分けて2つの種類に分類されます．1つはUNIXのmakeとそっくりに作ってあるものをベースにして，不必要な機能を削ったりさらに多くの機能をつけ加えたりしたものです．もう1つは，名前はmakeですが中身がまったく違うもの，というmakeです．

MS-DOSに付属のmakeは後者，MS-DOS用のCコンパイラに付属のmakeは前者のものが多いようです(MS-Cなどに付属のNMAKEなど)．

■ MAKEの効用

MAKEは複数のソースファイル，複数の実行ファイルからなる1つのシステムを作り上げる際に，コンパイル，リンク時の能率を上げるために考えられたソフトウェアツールです．

大きなアプリケーションになればなるほど，記述したプログラムが一度で動くということはほとんどありません．そのために何度かデバッグ作業を行うことになりますが，その際，最も面倒なことはコンパイル／リンクという作業の繰り返しです．あるファイルを修正したからといって，分割されたすべてのファイルのコンパイルとリンクを行っていたのでは時間のむだです．かといって，どのファイルをコンパイル／リンクするべきかを考えながら行うということは，大規模なシステムになればなるほど非常にやっかいです．また，こういった込み入った作業はバッチファイルで記述することはできません．この一連の作業を実に簡単に行ってくれるのが，MAKEというコマンドなのです．

MAKEは「MAKEFILE」と呼ばれる専用のテキストファイルに，コンパイルなどのオペレーションの手順とその関係を記述することにより，これまで人の手で行っていた再コンパイルすべきファイルの選択や実行を自動的に行わせることができます．つまり，MAKEはMAKEFILEという〝作業手順書〟をもとに，実行ファイルの作成にかかわるすべての作業を自動的に行うわけです．MAKEはな

にもC言語だけで使われるツールではなく，FORTRANやPascal，アセンブラなどのコンパイラ系言語で利用することができます．

下の**図7-1**にMAKEの概念を示しておきましょう．

MAKEはこの作業のよりどころとして，コンパイルやリンクによってできる成果物とそのもととなるファイルの時間関係を用います．これによって，MAKEが起動した時点で更新されたファイルのみを選択し，コンパイルやリンクを行うのです．

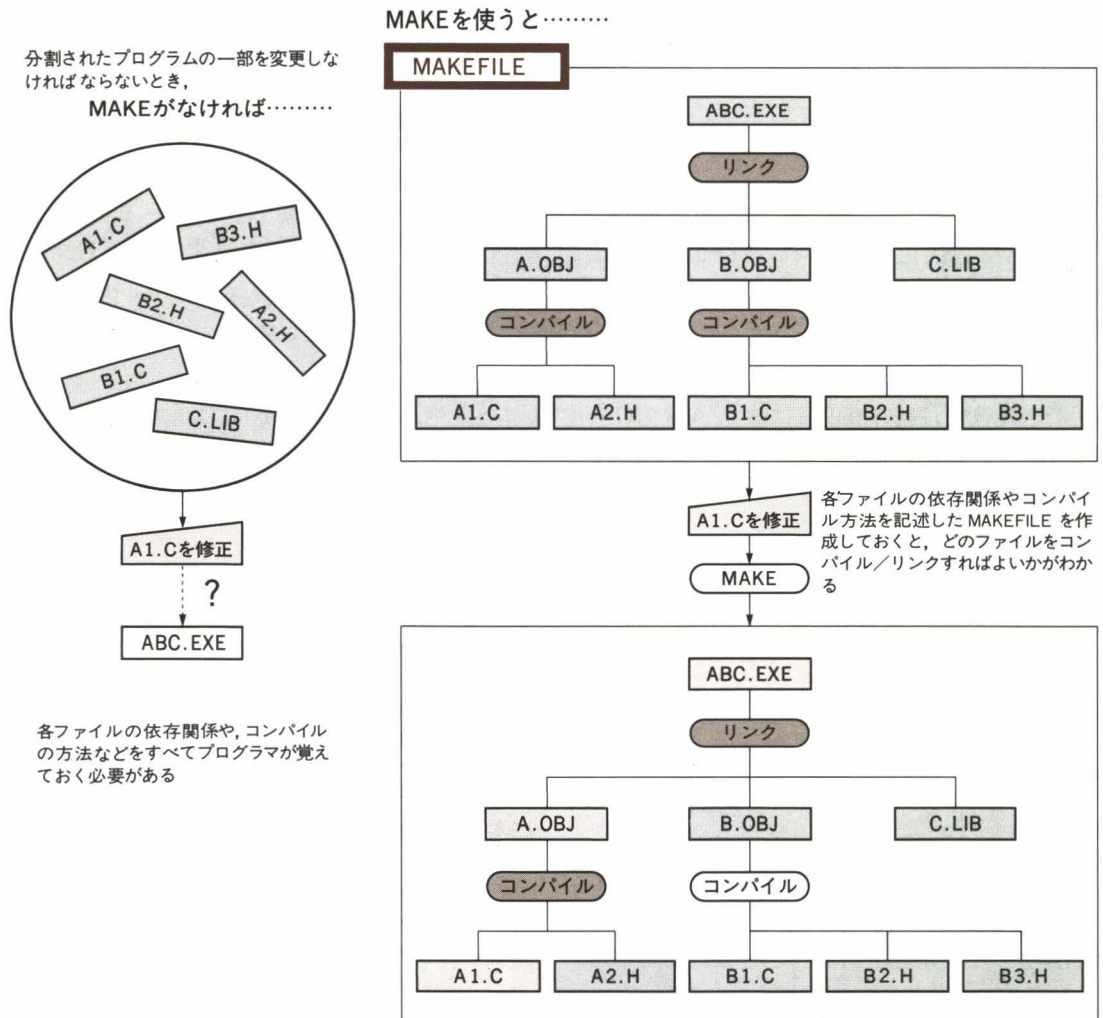
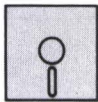
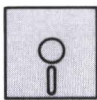



図7-1　MAKEの概念

図 7-2 に示すように，MAKE コマンドはファイルの日付と時間を調べ，コンパイルやリンクを実行するかどうかを判断します．ソースファイルとオブジェクトファイルの時間関係を調べれば，ソースファイルが更新されたかどうかがわかるので，図 7-2 の下の場合のみ，作業を行わせることが可能です．

MAKE の原理は簡単にいって以上のようなものですから，バックアップファイルの更新のようにファイルの時間関係が関与するファイル操作のすべてに応用が効きます．

〈通常の場合〉

	日付	時間
A1.C	90-10-17	19：20
A2.H	90-10-16	20：24
A.OBJ	90-10-17	19：32

MAKEFILEの中身

```
A.OBJ : A1.C A2.H
        CL -c -AL A1.C,A.OBJ ;
```

上のような記述があったとき，MAKEはA1.C, A2.H, A.OBJの作成日付，時間を調べる．左のような場合，A.OBJが最も新しいのでMAKEは何も行わない

〈A1.Cが修正された場合〉

	日付	時間
A1.C	90-10-24	23：14
A2.H	90-10-16	20：24
A.OBJ	90-10-17	19：32

修正されて，日付と時間が更新されている

MAKEFILEの中身

```
A.OBJ : A1.C A2.H
        CL -c -AL A1.C,A.OBJ ;
```

左のような場合，A1.C が，A.OBJ よりも新しいので，MAKE は，

CL -c -AL A1.C,A.OBJ ;

を実行する

図 7-2　MAKE の原理と動作

■ MS-DOSのMAKE

MS-DOS Ver3.1 以降に付属の MAKE は，現在ある MAKE のなかで最も簡単な構造をしているものの１つです*1. したがって，覚えて使いこなすまでにたいした労力を必要としないので，ぜひ使ってみることをお勧めします.

ここでは，MS-DOS の MAKE を例として，MAKEFILE の基本的な記述の仕方を見ていきましょう．MAKEFILE の基本的な構造を図 7-3 に示します.

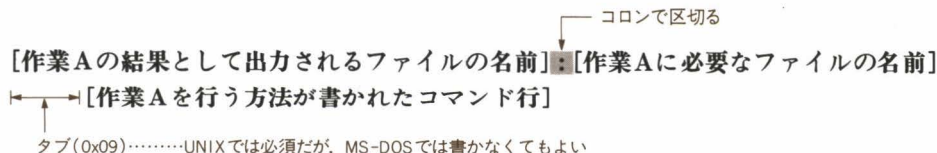



図 7-3　MAKEFILE の基本構造

この基本構造により，たとえば次のように記述します.

```
SAMPLE.OBJ : SAMPLE.C SAMPLE.H OTHER.H
        CL -AS -Od -Zd SAMPLE.C,SAMPLE.OBJ ;
```

最初の１行では，SAMPLE.OBJ というファイルは，SAMPLE.C，SAMPLE.H，OTHER.H という３つのファイルから作られていることが記述されています．次の１行はタブコードで１段つけてから，SAMPLE.OBJ の作り方が MS-DOS のコマンドの実行という形式で記述されます．この２つを使って，ある１つのファイルを作るための過程を記述します. この２行は MAKE コマンドで解釈されますが，次にその仕組みをお話ししましょう.

■ MAKEコマンドの動作の仕組み

MAKE コマンドは，最初の１行で記述されている「ファイル同士の関係」を使って〝その関係が時間どおりになっているか？〟を調べます．つまり，SAMPLE.OBJ がこれらのファイルのなかでいちばん新しければ，「ファイルの作られた時間関係は正常」ということになります．SAMPLE.OBJ は SAMPLE.C と SAMPLE.H と OTHER.H から作られたはずですから，このなかのファイルでいちばん新しいものは，SAMPLE.OBJ です.

*1　MS-DOS，マクロアセンブラ，C コンパイラに付属する MAKE には，いくつかバージョンがあり，その機能が違う．ここでは最も基本的な機能を持つものを想定して解説する.

SAMPLE.C，SAMPLE.H，OTHER.H から SAMPLE.OBJ が作られるのだから，
SAMPLE.OBJ がいちばん新しくなければならない

さて，ではこのファイル同士の時間関係がこういった「正常な関係でない」場合とはどのようなときでしょうか？　これは，SAMPLE.C か SAMPLE.H か OTHER.H のいずれかが，なんらかの原因で SAMPLE.OBJ よりも新しくなってしまったということです．もちろん，こういうことが起こる最大の理由は，ソースファイルの修正によるものです．つまり，この場合はコンパイルをしなおさなければなりません．さもないと，コンパイルされた結果としての SAMPLE.OBJ は，SAMPLE.C をコンパイルした結果とは違ったものになってしまいます．

ここで 2 行目が参照され，MAKE によって実行されます．つまり，1 行目の関係を正常にすべく，再コンパイルが行われます．

SAMPLE.OBJ がいちばん新しくない場合は，2 行目が参照され，
SAMPLE.OBJ を作り直す

MAKE はこの 2 行の情報の集まりをあるファイルに記述しておき，その情報を読み，それに基づいてコンパイルを実行するかどうかを決定します．前述したように，この 2 行の情報は 2 行ずつが 1 つの意味を持つものとして解釈されるので，たとえば実際には**図 7-4** のように複数の 2 行の情報がファイルに書かれます．

```
1: SAMPLE.OBJ : SAMPLE.C SAMPLE.H OTHER.H
2:          CL -c -AL SAMPLE.C,SAMPLE.OBJ;
3:
4: SAMPLE2.OBJ : SAMPLE2.C STDIO.H
5:          CL -c -AL SAMPLE2.C,SAMPLE2.OBJ;
6:
7: SAMP.EXE : SAMPLE.OBJ SAMPLE2.OBJ
8:          LINK SAMPLE.OBJ+SAMPLE2.OBJ,SAMP.EXE,NUL/NOD;
```


注：上記のようにコンパイルとリンクを別々に行っていることに意味がある
これを，

```
SAMP.EXE : SAMPLE.C SAMPLE2.C……
        CL -FeSAMP *.C
```

と書いてはmake を使う意味がない

図 7-4　MAKEFILE の実例

ここでは3つのコンパイルやリンクの情報が書かれています.これが順に解釈されていくわけです.後述するUNIXのmakeと異なり，かならず記述の順に解釈されますから，たとえば図7-4では「SAMP.EXE」を「SAMPLE.OBJ」の前に書くことはできません.

MAKEでは，こういった2行の情報が3行になっても4行になってもかまいません．つまり，最初の1行(時間関係を表す)のあとに，いくつもの実行コマンドをつけ加えることができます．ただし，この場合は2行目以下のコマンド行のあいだはあけてはいけません．また各まとまりの情報同士は，1行以上のブランク行で区切られている必要があります．そうしないと，MAKEコマンドは構文エラーとみなしてしまいます.

このコンパイル情報の記述してあるファイルを，たとえば「MAKEFILE.MSC」とすると，MAKEコマンドは以下のように起動します.

```
A>MAKE  MAKEFILE.MSC
```

このようにMAKEというツールは，MS-DOSのファイルのディレクトリ情報のうち，そのファイルが作られた時間情報(タイムスタンプ)をもとに，ファイルの時間関係の比較を行い，指定されたコマンドを自動実行していきます(MS-DOSのタイムスタンプを取り出す例題として，第4章ではFINDFを取り上げた).

一度MAKEFILEを記述しておけば，上記のコマンドを実行するだけでコンパイルやリンクの作業がすんでしまいます.簡単なプログラムで再コンパイルの必要がほとんどないものならば，バッチファイルで記述しておくという手もありますが，複数のプログラムに分割され何度もデバッグ作業を要するプログラムでは，MAKEを使う方が圧倒的に便利です.

■ MAKEコマンドの応用

MAKEコマンドはなにもコンパイルのみに使うわけではありません．たとえばファイルのバックアップなどでは，図7-5のようなMAKEFILEを作ることができるでしょう.

MAKEというツールはたいへん便利なものですが，MS-DOSなどの比較的メモリ領域が小さく，仮想メモリ機構も持っていないオペレーティングシステムで動かすと，どうしてもメモリ不足のことを考えながら使うことになります．とくに最近のコンパイラは巨大化しており，日本語フロントエンドプロセッサなどの大きな常駐プログラムと併せて使われる場合などは，コンパイラそのものは単独で動いてもMAKEを使うと動かなくなる場合もないとはいえません.

```
    ┌──→ドライブ名の指定は左辺と右辺の区切り記号“:”と区別がつかないので不可*
 1: backupf : file.a file.b file.c file.d file.e……バックアップされるファイル
 2:         command /c if NOT EXIST backupf mkdir b:¥bakupd
 3:         └──→MS-DOSの内部コマンドを実行するときはCOMMAND.COMを呼ぶ
 4: backupf : file.a
 5:         command /c copy file.a b:¥backupd
 6:
 7: backupf : file.b
 8:         command /c copy file.b b:¥backupd
 9:
10: backupf : file.c
11:         command /c copy file.c b:¥backupd
12:
13: backupf : file.d
14:         command /c copy file.d b:¥backupd
15:
16: backupf : file.e
17:         command /c copy file.e b:¥backupd
18:
19:         :
20:         command /c echo BACKUP >backupf
                      └── この1行はMAKEが実行されるたびに常に実行され，
                          backupfは更新される
```

*MAKEのバージョンによって異なり，ドライブ指定が可能なものやMS-DOSの内部コマンドを外部コマンドと同様に扱えるものもある

図7-5　バックアップ用のMAKEFILE

■ UNIXのmake

さて，このMAKEというツールは，UNIXのものが最初であるということを冒頭でも述べました．その UNIXのmakeは，前項に紹介したMS-DOSのMAKEと基本的には同じですが，考え方が少し異なります．以下にUNIXのmakeについて解説しましょう．なお，MS-DOSでもUNIXのmakeとほぼコンパチビリティを保ったものがいくつかあります(MS-C　ver5.1，QuickCなどに付属のMAKE，CプログラムブックII［アスキー出版局発行］に掲載されたMAKEなど)．

UNIXではとくにファイルを指定せず，ただ「make」とだけ打つとカレントディレクトリの「makefile」または「Makefile」という名前のファイルを，makeの規則を記述してあるファイルとみなしてmakeコマンドの実行が行われます(ファイル名を指定することもできる)．

UNIXのmakeは，MS-DOSのMAKEと基本的な規則の記述は同じですが，その組み合わせ方が異なっています．UNIXでのmakefileは，ファイル全体の構造が問題とされ，MS-DOSのように記述の順序はあまり問題になりません．それはたとえば次ページの図7-6と図7-7のような記述の違いをみると明らかでしょう．

```
1: test.obj : test.c test.h
2:         cl -c -AL test.c;
3:
4: test1.obj : test1.c
5:         cl -c -AL test1.c;
6:
7: test.exe : test.obj test1.obj
8:         link test+test1,test.exe,nul/NOD
```


図7-6　MS-DOSのMAKEFILEの記述

```
1: test.exe : test.obj test1.obj
2:         link test+test1,test.exe,nul/NOD;
3:
4: test.obj : test.c test.h
5:         cl -c -AL test.c;
6:
7: test1.obj : test1.c
8:         cl -c -AL test1.c;
```


図7-7　MAKEFILE(図7-6)をUNIX流に記述した場合

すなわち，MS-DOSのMAKEはそのファイルのタイムスタンプの検証と，その結果による実行文の検証，実行の「順序」が書かれているのに対して，UNIXのmakeは成果物としてファイルができるまでのコンパイル作業全体の「構造」が記述してあるのだといってもよいでしょう．そのため，UNIXのmakeでは，まず始めに最終的にどのようなファイルが作成されるのかをかならず記述する必要があります．

さらに，UNIXのCコンパイラはオプションなどが共通で，みな同じフォーマットのコマンド行でコンパイルができるので，フラグ類のみを指定するだけでコンパイルはmakeが自動的にやってくれます．

図7-8にUNIXのMakefileの記述例を示します．

```
 1: #
 2: #       genka make          } #で始まる行はコメント行
 3: #
 4: CFLAGS = -O -s
 5: LIBDIR = /usr/lib
 6: OBJS   = genka.o oss.c      } マクロ定義
 7: SRCS   = genka.c oss.c
 8:
 9: genka : $(OBJS)             } リンクの手順しか書いていないが，コンパイルも実行する.
10:         cc $(OBJS) -o genka } コンパイルはデフォルトの規則として定義されている
11:
12: setup:
13:         cp ./genka /usr/genka/bin/           } "make setup"とすると，この
14:         chmod 755 /usr/genka/bin/genka       } ブロックが実行される
15:
16: clean :
17:         rm -f ./$(OBJS)
18:         rm -f ./genka
19:
20: print:
21:         pr $(SRCS) | lpr
```

図 7-8　UNIX の Makefile の例

このように，UNIX の make は**マクロ**が使用できるのも大きな特徴です．マクロを使うと，たとえば以下のような指定ができます．

<マクロの定義>

OBJS = test.obj test1.obj test2.obj　……マクロ名＝定義するファイル名，オプションなど

<定義したマクロの参照>

test.exe ： $(OBJS)　……$(マクロ名)で参照する

定義するときには定義文字を左辺に，定義される内容を右辺に置き，それを「＝」で結びます．そのマクロを使う場合は，定義された文字をかっこでくくり，その最初に「＄」の文字をつけて使うわけです．これを利用すれば，同じような Makefile を作るときに，マクロ定義の内容だけを書き換えればよいことになります．

このマクロにはコンパイル時にコンパイル・オプションを指定するためなど，最初から役目の決まっているマクロ名があり，これはユーザーが使うマクロ名としては避けなければなりません(次ページの**表 7-1** 参照)．もし，同じ名前で定義すると，デフォルトで定義してある規則が消え，記述した規則が新たに定義されます．

マクロ名	定義の内容
MAKE	makeコマンド自身が定義されている
AR	ar(アーカイブ)コマンドが定義されている
ARFLAGS	arのフラグを定義する
AS	as(アセンブラ)コマンドが定義されている
ASFLAGS	asのフラグを定義する
CC	cc(Cコンパイラ)コマンドが定義されている
CFLAGS	ccのフラグを定義する
F77	FORTRANコンパイラが定義されている
F77FLAGS	FORTRANコンパイラのフラグを定義する
GET	SCCSで使うgetコマンドが定義されている
GFLAGS	getコマンドのフラグが定義されている
LEX	lexコマンドが定義されている
LFLAGS	lexのフラグを定義する
LD	ローダのコマンドが定義されている
LDFLAGS	ローダのフラグを定義する
YACC	yaccコマンドが定義されている
YFLAGS	yaccのフラグを定義する

表7-1　予約されているマクロ一覧

また make は，Makefile 上の任意の場所から実行することができます．たとえば先の図7-8で，

```
%make  setup
```

とすると，Makefile のなかの「setup：」というラベルで記述された内容のみを実行します．

これを使うと，make でコンパイルが終了した後，コンパイルされた結果をある特定のディレクトリにコピーするといった一連の操作が同一の Makefile 上に書けるようになるわけです．図7-8では setup のほかに，clean や print などのラベルと実行文をつけて，プログラム作成に付随する各種の作業を make コマンドで実行できるようにしています．

さらに make では

```
%make  -f  newmakefile
```

とすると，Makefile の代わりに newmakefile という名前のファイルを Makefile として扱ってくれるようになります．これを使うと Makefile の構造化が可能になります．つまり，同一のディレクトリでいくつかのシステムを作らなければならない場合などに，いくつもの名前の違う Makefile を作り，システムごとに make が make を呼ぶような Makefile を作成します．

UNIXの標準的なmakeには，このほか**表7-2**のようなオプションがあります．

オプション	機　　能
-i	makeがコマンドを実行しているときのエラーを無視する(通常はmakeが止まってしまう)
-s	実行中の表示を行わない(バックグラウンドでmakeを動かしたいときなどに使う)
-r	デフォルトの規則(コンパイル等)を実行しない
-n	コマンドを実行せずに表示のみを行う(Makefileのデバッグに使う)
-t	make実行後に更新されたターゲットファイルにtouchコマンドを実行する
-q	ターゲットの更新が行われればステータス0を返す(シェルスプリクト中でmakeを使う場合に用いることができる)
-p	マクロ定義とターゲットについての記述をすべて画面に表示する
-k	実行中に誤りがあると，それまでの実行結果をすべて破棄する
-e	Makefile中の環境変数への代入を変更する
-f	このオプションに続くファイル名のファイルをMakefileとして実行する

表7-2　make起動時のオプション一覧

makeはこれらのオプションをMAKEFLAGSという環境変数から参照し，makeの起動に当たっては，これらのオプションを取り込んで実行することができます．

Makefileも大きくなると数10Kバイトの大きさになることがあり，「-n」のようなフラグで構文チェックをしないとうまく動かないなどということもあるでしょう．これらのフラグはなかなか有効に使えるものです．

「Makefileを作る」ということもプログラムを作る作業のうちに入ってくるわけですが，その効果は計り知れません．とくに，多人数での大規模なシステム構築になればなるほど，これらのmakeの効果ははっきりしてきます．しかし，そのぶんMakefileを作る作業も膨大なものになってくることはいうまでもありません．そこで，4.2BSD以降のバークレイ版UNIXなどは，「mkmf」というツールが用意されており，Makefileを自動的に生成してくれます．

なんだかロボット工場でロボットがロボットを作るような話ですが，やはり便利であることはいうまでもありません．しかし，こういったツールではきめの細かい制御ができないという短所もあります．

7.2 DEBUG, SYMDEB ― デバッグ用ツール ―

デバッグを行う場合の基本は，なんといっても「バグのないプログラムを作る」という当たり前のことになってしまいます．そのためには，せめて全体のフローなどは紙に書き，その上で机上デバッグを慎重に行います．プログラムを組む作業に慣れてくると，すぐにコンピュータに向かってプログラムを打ち込むことが多くなりますが，せめてシステムの設計は机上でのデバッグを行ってください．

きちんと設計したプログラムはなんということもなく，すんなりと動いてしまうのが普通のはずですが，実際はスペルミスなどの小さいバグから，システム設計時点でのミスのような大きなバグまで，さまざまなバグに悩まされるのが普通です．この部分の作業効率がシステム全体の構築の時間を決定してしまうことも多くあります．

■ デバッガの機能

デバッガはアセンブラやＣ言語でのプログラム開発には欠かせない存在ですが，最近のように，プログラムがアセンブラで追えるほど単純なものでなくなり，コンパイル速度の向上により再コンパイルがあまり煩わしくなくなってきた現状では，だんだんとその使われ方も変わってきているようです．

もともと，デバッガはアセンブラレベルでのプログラム開発を支援する目的で作られたものです．したがって，ここで紹介するMS-DOSに付属のDEBUGやSYMDEBコマンドではアセンブラの知識が必要になってきます．

また，デバッガではどんなことをしてはいけないのか，どういうプログラムにデバッガが使えないのかといったことは，ハードウェアも含めたコンピュータの総合的な知識を必要とします．つまり，Ｃ言語の知識だけではデバッガを使ったデバッグは行えないということです．たとえば，最も原始的なデバッガ（最も構造が簡単なデバッガ）は，Ｃ言語で書かれたプログラムそれ自体をデバッグするのではなく，コンパイルした結果の機械語のプログラムをデバッグすることになるので，機械語（アセンブラ）の知識が最低限必要になります．

デバッガは機械語で動いているプログラムに対してデバッグを行います．デバッグという行為は，プログラムを実行させ，その結果としてのデータ値を得て，それが正しいかどうかを判断するということなので，デバッガには以下のような機能が備わっています．

1. **プログラムの実行を任意に制御する**
   - ・プログラムを１ステップ（１機械語命令）ごとに実行して止める
   - ・決められたステップまでプログラムを実行して止める

2. メモリの操作をする

・必要な任意のメモリの内容を読む

・必要な任意のメモリの内容を書き換える

以上がデバッガの基本機能ですが，この基本機能を組み合わせると，拡張機能として，

3. 必要なメモリの内容を表示しながらプログラムを実行する

4. 実行中に任意の場所でプログラムを止める

といった機能が使えるようになります．

さらに，C言語などの高級言語のデバッグにも使えるように，データベース機能をつけて，〝ソースコードのどの行が機械語のどのステップに対応するのか〟といった情報の管理が行えるデバッガもあります．またこのようなデバッガでは，変数がメモリをどのように使っているのかという情報もデータベースで管理します．つまり，

5. 高級言語やアセンブラのソースコードレベルでの機械語デバッグを行う

6. メモリをソースコード中で使われている名前で管理する

ということが可能になります．

後述するSYMDEBといったデバッガは，プログラムやデータ領域にソースコードでつけられた名前と，実際のメモリアドレスの対応表を作っておき，その名前のデータベースから必要なメモリやプログラムのメモリ上での位置を捜して，人間がデバッグしやすいように作ってあるわけです．この機能がないいちばん原始的なデバッガでは，これらの〝メモリ〟対〝名前〟の対応表を見ながらデバッグを行う必要があります．

しかしデバッガというのは，あくまでターゲットとなるプログラムの実行を制御し，メモリの内容を覗いたり書き換えたりすることが主な仕事であり，ソースコードと機械語の対応や変数名と実際のメモリエリアの対応などは，便利な付加機能にすぎません．

■ 最も基礎的なデバッグの実際

デバッグの基本機能がわかったところで，実際のデバッグ手順を追ってみることにしましょう．ここでは，最も基礎的なデバッガの例として，MS-DOS Ver3.1に付属のDEBUGコマンドを使ってみます．これはMS-DOS Ver3.3以降に付属のSYMDEBでも同様のことが(コマンドを含んでいる)できます．

たとえば，プログラムを途中のfunc()という関数から実行させ，それが終わったら，またデバッガに制御を戻すといった場合を考えてみましょう．

①プログラムをリンクする際に「/MAP」オプション(MS-DOSのLINK.EXEの場合)をつけ，マップファイルを出力させ，それをいつでも見ることができる状態－すなわちプリントアウトして，手元に置いておきます．

②デバッガを立ち上げ，ターゲットプログラムがデバッガの下で動くようにします．

③次にfunc関数の機械語での位置を捜します．
　Cコンパイラによっても違いますが，MS-Cの場合であれば，C言語で書かれたfunc関数は，アセンブラではその関数名の頭に「_」(アンダースコア)のついた「_func」というラベルから始まるアセンブラプログラムとして解釈されています．そこで，マップで「_func」というラベルを捜します．func関数はそのアドレスから始まる機械語になっているはずです．

④「_func」は「0000：0020」というアドレスから始まっていたとします．
　EXE形式のファイルは，MS-DOSによってロードされるアドレスが決まりますから，DEBUG起動時のCSレジスタの値をマップで示されたアドレスに加算する必要があります．このCSの値を14DFとすると，func関数はメモリ上では「14DF：0020」から始まっていることになります．
　プログラムが動き始めてからこのfunc関数に至るまでに，ひょっとするといろいろなグローバル変数などが書き換えられていて，その変数がfunc関数に影響を与えるかもしれないので，すぐにこの場所からプログラムを起動させるのはよくありません．そこで，このfunc関数の入り口のアドレス，すなわち「14DF：0020」というアドレスにプログラムが来たら，実行を止めてデバッガに制御が戻るようにします．
　たえばDEBUGでは，

```
-G  14DF:0020
```

とします．プログラムはあっという間にここまで実行され，再びデバッガに制御が戻ります．プログラムの実行が終わるとレジスタの値が以下のように表示されるので，「14DF：0020」の直前まで実行されたことがわかります．

```
CS = 14DF     IP = 0020
```

⑤ここからはプログラムを1(機械語)命令ずつ実行させてみます．

```
-T
```

「T」と入力すると1ステップだけ実行し，プログラムはデバッガに制御を戻します．もちろん，もう1回「T」を実行すると同じように次の1ステップを実行します．

⑥あるところで「hensuu」という名前のグローバル変数の値を見る必要があったとします.

またマップを参照し，前と同じように「_hensuu」のアドレスを求めます*2. たとえばそのアドレスが「0141：0036」であったとすると，前述の関数のアドレスと同様に「(デバッガ起動時のCSの値)＋0141h＝1620h」が「_hensuu」のセグメントアドレスなので,

```
-D 1620:0036
```

とすると，そのメモリ番地からのダンプが取れますから，メモリの内容がわかります.

⑦次に，同じメモリの内容を書き換える場合は,

```
-E 1620:0036 FF
```

とすると，その番地にFFという値が書かれます.

デバッガの基本操作はこういったところです. 次ページの図7-9にこれまでの実行過程を示してみます.

これらの操作のうち，なんといっても面倒なのがマップの参照です. そこで，これらのマップの参照をデバッガの機能として組み込んでしまったものが，以降で紹介するシンボリックデバッガで，メモリ上の番地の参照をデバッガプログラムが自動的にやってくれます. たとえば，Dコマンドでメモリ内容をダンプする場合は,

```
-D _hensuu
```

とすることによってデバッガが勝手に「_hensuu」のアドレスをマップファイルから読み込んで計算し，ダンプを実行します.

また,最近のツールでは,高級言語で書かれたソースファイルと機械語プログラムの対応データベースをもとに，デバッガプログラムが自動的にソースファイルの行番号情報や変数情報を管理し，ソースコードの表示されているスクリーンを見ながらデバッグを行うことなどが可能です.

*2 マップファイルにはグローバル変数のアドレスのみが書き出される. ローカル変数はスタック領域にストアされるため，マップファイルからはそのアドレスを知ることはできない. ただし実行時にスタックを見て，調べることはできる.

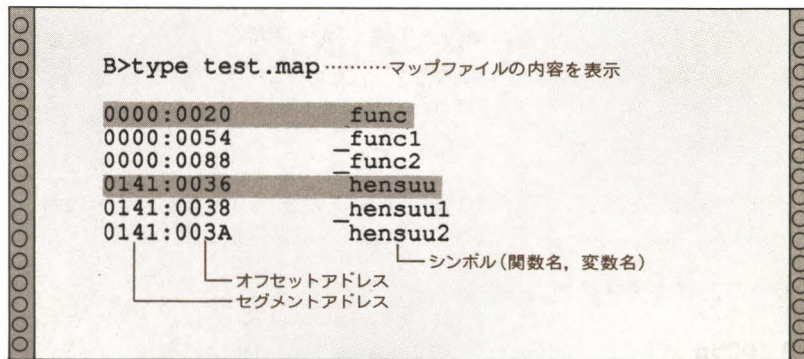
```
B>type test.map ………マップファイルの内容を表示

0000:0020          _func
0000:0054          _func1
0000:0088          _func2
0141:0036          _hensuu
0141:0038          _hensuu1
0141:003A          _hensuu2
                      └─シンボル(関数名, 変数名)
         └─オフセットアドレス
    └─セグメントアドレス
```

```
B>debug test.exe
                               ┌─プログラムのロードアドレス
-r
AX=0000  BX=0000  CX=1784  DX=0000  SP=0800  BP=0000  SI=0000  DI=0000
DS=14CF  ES=14CF  SS=169A  CS=14DF  IP=00BC   NV UP EI PL NZ NA PO NC
14DF:00BC B430          MOV     AH,30
-g 14DF:20 ………………………_funcのセグメントはマップファイルのセグメントにCSを加算したものになる

AX=0000  BX=0F8E  CX=0000  DX=0080  SP=0F82  BP=0F84  SI=006F  DI=101D
DS=1620  ES=1620  SS=1620  CS=14DF  IP=0020   NV UP EI PL ZR NA PE NC
14DF:0020 55            PUSH    BP
-t ………………………1ステップ実行してみる

AX=0000  BX=0F8E  CX=0000  DX=0080  SP=0F80  BP=0F80  SI=006F  DI=101D
DS=1620  ES=1620  SS=1620  CS=14DF  IP=0023   NV UP EI PL ZR NA PE NC
14DF:0023 C70636000100  MOV     WORD PTR [0036],0001              DS:0036=0000
-d 1620:36,4F ……_hensuuのセグメントはマップファイルのアドレスにCSを加算したもの(CS+141H=1620H)となる
1620:0030                    00 00-00 00 00 00 25 64 0A 00         ......%d..
1620:0040  25 64 0A 00 25 64 0A 00-25 64 0A 00 25 64 0A 00   %d..%d..%d..%d..
-e 1620:36 FF ………………………_hensuuの値を変える
-d 1620:36,4F
1620:0030                    FF 00-00 00 00 00 25 64 0A 00         ......%d..
1620:0040  25 64 0A 00 25 64 0A 00-25 64 0A 00 25 64 0A 00   %d..%d..%d..%d..
-q

B>
```

図7-9　DEBUGの基本的な使い方

■ デバッガの基本構造

こういったデバッガの基本機能は，どのように実現されているのでしょうか？

メモリの内容を見たり書き換えたりすることは，通常のプログラムで何度もやっていることなので，説明の必要はないでしょう．プログラムを任意の場所で止めるということは，一見するとかなり難しいことをやっているようですが，実はたいしたことではありません．

デバッガはプログラムを止めたい場所の命令コードを勝手に書き換えて，そこに(8086･CPUならば)割り込み命令のINTを置き，そのINTが実行されるとデバッガが起動するようにしてあるだけなのです．簡単な方法では単にJMP命令を置いて，そのジャンプした先にはデバッガのプログラムがあるといったことでもかまいません．

つまり，デバッガは必要に応じてターゲットとなるプログラムを勝手に書き換えて実行の制御をします．もちろん，デバッガは書き換えた場所とその内容を記憶しておき，その場所(**ブレークポイント**という)が必要なくなった時点で，その命令を取り去るわけです．

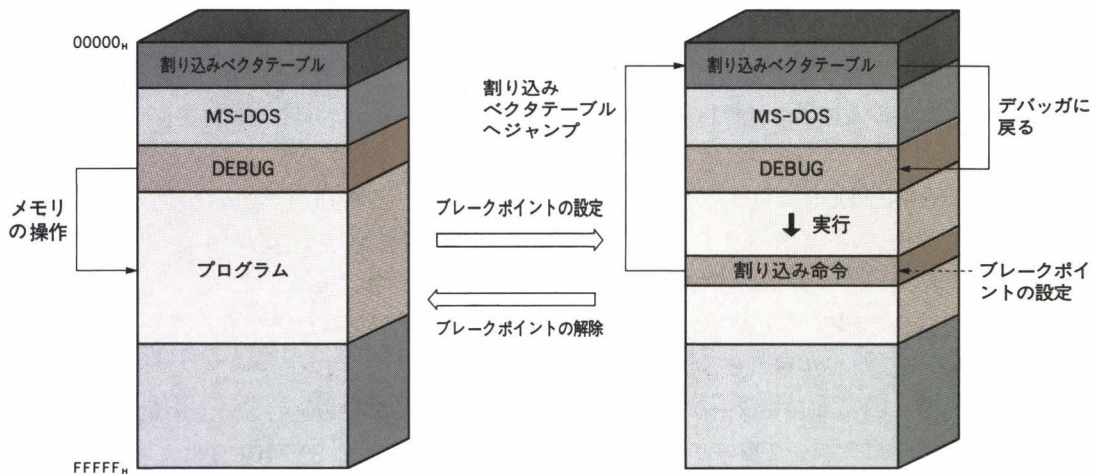



図7-10 デバッガの基本構造

前述のような仕組みでデバッガは動いているため，次のようなプログラムは基本的にデバッグできません．

- プログラムのコードを常にチェックしているようなプログラム
- デバッガと同じINTの番号を取り合うプログラム
- デバッガとターゲットプログラムの大きさを合わせると，大きすぎて実行不可能になってしまうプログラム
- デバッガが使うメモリエリアを勝手に使ってしまうプログラム
- デバッガで使っているキーボードや画面などの入出力にかかわる部分を勝手に書き換えてしまうプログラム
- デバッグの対象プログラムがROM化されているプログラム

こういったプログラムをどうしてもデバッグしたい場合は，デバッガではなく，ICE(In Circuit Emulator)というハードウェアを用いることになります．

■ SYMDEBを使ったデバッグ

前述したように，DEBUGコマンドでは，コンパイル，リンク時にマップ情報をリンカなどのオプションで出力させ，そのプリントアウトを参照しつつ，実際のメモリ上の機械語プログラムをデバッグしました(実際には，そのあとで機械語プログラムに相当するソースファイルの修正を行う必要がある)．

この同じ作業をMS-DOSに付属のSYMDEBコマンドで行うと，機械語で扱うにしても，そのアドレスラベルや変数ラベル*3でデバッグできるので，たいへん便利です．また**図7-11**に示すように，行番号情報を含んだマップファイルを作成すると(コンパイラおよびリンカのオプションで指定する)，ソースプログラム・レベルでのデバッグが可能(ソースプログラムの何行目という指定ができる)になります．SYMDEBでラベル参照を行うには，MAPSYMコマンドを用いてマップファイルからシンボルファイルを作成し，起動時にターゲットプログラムといっしょに読み込むという手順をとります．

以下の**リスト7-1**にデバッグ用のサンプルプログラムを，図7-11にSYMDEBによるデバッグの過程を示します．

---

*3　変数ラベルで参照できるのは，グローバル変数のみ．

```
 1: /*
 2:  *  linear - solve a linear equation
 3:  *
 4:  *    Written by Katsufumi Fukunishi
 5:  *
 6:  *    Copyright (C) 1987 by Katsufumi Fukunishi
 7:  *
 8:  *    Created    Nov. 16, 1987
 9:  *    Modifyed   Sep. 25, 1990   by Nishi Katsuhiro
10:  */
11:
12: #include <stdio.h>
13: #include <math.h>
14:
15: #define DM  10 ……………………計算できる次元の最大値
16:
17: main()
18: {
19:     int     i, j, n;
20:     double  m[DM][DM+1];
21:
22:     fscanf( stdin, "%d", &n );                      ┐
23:     for ( i = 0; i < n; i++ )                       │
24:         for ( j = 0; j <= n; j++ )                  │データの読み込み
25:             fscanf( stdin, "%lf", &(m[i][j]) );     ┘
26:
27:     for ( i = 0; i < n; i++ ) {
28:         for ( j = 0; j < n; j++ )                         ┐
29:             fprintf( stdout, "%+lg x%d ", m[i][j], j );   │方程式の表示
30:         fprintf( stdout, " = %lg¥n", m[i][n] );           │
31:     }                                                     ┘
32:
33:     gauss( m, n );
34:
35:     for ( i = 0; i < n; i++ )                         ┐
36:         printf( "x%d = %.20lg¥n", i, m[i][n]);        ┘解の表示
37: }
38:
39: gauss(double m[DM][DM+1], int n)
40: {
41:     int i,j,k,max;
42:     double a;
43:
44:     for ( k = 0; k < n-1; k++ ) {
45:         max = k;
46:         for ( i = k+1; i < n; i++ )                               ┐
47:             if ( fabs(m[i][k]) > fabs(m[max][k]) )                │max( | m[i][k] | ;k<i<n)
48:                 max = i;                                          ┘を求める
49:         if ( max != k )
50:             for ( j = k; j <= n; j++ ) {                          ┐
51:                 a = m[k][j];                                      │
52:                 m[k][j] = m[max][j];                              │k行とmax行を入れ換える
53:                 m[max][j] = a;                                    │
54:             }                                                     ┘
```

```
55:         for ( i = k+1; i < n; i++ ) {
56:             a = m[i][k] / m[k][k];
57:             for ( j = k+1; j <= n; j++ )
58:                 m[i][j] += a * m[k][j];
59:         }
60:     }
61:     for ( k = n-1; k >= 0; k-- ) {
62:         m[k][n] /= m[k][k];
63:         for ( i = k-1; i >= 0; i-- )
64:             m[i][n] -= m[i][k] * m[k][n];
65:     }
66: }
```

（55〜65行：行列計算）

リスト7-1　LINEAR.C

```
A>msc -Zd -Od linear;
Microsoft (R) C Compiler  Version 4.00
Copyright (C) Microsoft Corp 1984, 1985, 1986.  All rights reserved.
A>link linear/NOI/M/LI;
Microsoft (R) Overlay Linker  Version 3.51
Copyright (C) Microsoft Corp 1983, 1984, 1985, 1986.  All rights reserved.
```

注記：/NOI…大文字と小文字の区別、/M…マップファイルの生成、/LI…マップファイルに行番号とアドレスを書き込む

```
B>cl -Zd -Od -Fm linear.c
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

linear.c

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: LINEAR.OBJ /LI
Run File [LINEAR.EXE]: LINEAR.EXE /NOI
List File [D:LINEAR.MAP]: LINEAR.MAP /M
Libraries [.LIB]:

B>mapsym linear
Microsoft Symbol File Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
        Program entry point at 0000:0430

B>type linear.dat
2
1.0  2.0  1.0
2.0  1.0  2.0
```

注記：-Zd…行番号情報を含むデバッグ情報をオブジェクトファイルに書き込む、-Od…最適化を禁止する、mapsym…マップファイル(.map)からシンボルファイル(.sym)を生成

```
B>linear < linear.dat⏎
+1 x0 +2 x1  = 1
+2 x0 +1 x1  = 2
x0 = 0.6 }
x1 = 0.8 }……… (X0=1, X1=0) になるはず

B>symdeb linear.sym linear.exe⏎ ……………… シンボルファイルと実行ファイルを指定してSYMDEBを起動する
Microsoft Symbolic Debug Utility
Version 3.01
(C)Copyright Microsoft Corp 1984, 1985
Processor is [80286]
-{linear.dat⏎ ……………… linear.datからデータを入力する
-s+ ……………… ソースレベルでのデバッグを行う
-g _main⏎ ……………… "_main"にブレークポイントを設定してプログラムを実行する
linear.c                関数名での実行の制御が可能
18:     {
-g _gauss⏎
+1 x0 +2 x1  = 1
+2 x0 +1 x1  = 2
40:     {
-v⏎ ……………… ソースラインを表示する
41:         int i,j,k,max;
42:         double a;
43:
44:         for ( k = 0; k < n-1; k++ ) {
45:             max = k;
46:             for ( i = k+1; i < n; i++ )
47:                 if ( fabs(m[i][k]) > fabs(m[max][k]) )
48:                     max = i;
-v⏎
49:             if ( max != k )
50:                 for ( j = k; j <= n; j++ ) {
51:                     a = m[k][j];
52:                     m[k][j] = m[max][j];
53:                     m[max][j] = a;
54:                 }
55:             for ( i = k+1; i < n; i++ ) {
56:                 a = m[i][k] / m[k][k];
-v⏎
57:                 for ( j = k+1; j <= n; j++ )
58:                     m[i][j] += a * m[k][j];
59:             }
60:         }
61:         for ( k = n-1; k >= 0; k-- ) {
62:             m[k][n] /= m[k][k];
63:             for ( i = k-1; i >= 0; i-- )
64:                 m[i][n] -= m[i][k] * m[k][n];
-g .55 ……… 55行目にブレークポイントを設定してプログラムを実行する. ソースファイルの行番号による実行の制御が可能
55:             for ( i = k+1; i < n; i++ ) {
-u .51⏎ ……………… 51行目から逆アセンブルする
51:
```

```
2599:025F B85800         MOV    AX,0058 ......m[i]とm[i+1]のアドレスの差
2599:0262 F76EF0         IMUL   Word Ptr       sizeof(double)×(DM+1)
2599:0265 8BD8           MOV    BX,AX
2599:0267 8B46F2         MOV    AX,[BP-0E]
2599:026A B103           MOV    CL,03
2599:026C D3E0           SHL    AX,CL
2599:026E 03D8           ADD    BX,AX
-u ......続きを逆アセンブルする
2599:0270 8B7604         MOV    SI,[BP+04] ......mのアドレス(ローカル変数のアドレス)
2599:0273 8D7EF8         LEA    DI,[BP-08]
2599:0276 8D30           LEA    SI,[BX+SI]
2599:0278 16             PUSH   SS
2599:0279 07             POP    ES
2599:027A A5             MOVSW
2599:027B A5             MOVSW
2599:027C A5             MOVSW
-dw bp+4 bp+5 ......bp＋4番地の内容をワード型で表示する    } ローカル変数はラベル参照ができない
2B5F:10E6  10F0 ......mのアドレス(m[0][0]のアドレス)        }
-dl 10f0 1100 ......dce番地からdde番地を倍精度実数型で表示する
2B5F:10F0  00 00 00 00 00 00 00 40  +0.2E+1......m[0][0]
2B5F:10F8  00 00 00 00 00 00 F0 3F  +0.1E+1......m[0][1]
2B5F:1100  00 00 00 00 00 00 00 40  +0.2E+1......m[0][2]
-h 10f0 58 ......m[0][0]とm[1][0]のアドレスの差
             ......0xdecと0x58の和と差を求める
1148  1098
-dl 1148 1158 ......m[1][0]のアドレス
2B5F:1148  00 00 00 00 00 00 F0 3F  +0.1E+1......m[1][0]
2B5F:1150  00 00 00 00 00 00 00 40  +0.2E+1......m[1][1]
2B5F:1158  00 00 00 00 00 00 F0 3F  +0.1E+1......m[1][2]
-v
51:                     a = m[k][j];
52:                     m[k][j] = m[max][j];
53:                     m[max][j] = a;
54:                 }
55:             for ( i = k+1; i < n; i++ ) {
56:                 a = m[i][k] / m[k][k];
57:                 for ( j = k+1; j <= n; j++ )
58:                     m[i][j] += a * m[k][j];
-v
59:             }
60:         }
61:         for ( k = n-1; k >= 0; k-- ) {
62:             m[k][n] /= m[k][k];
63:             for ( i = k-1; i >= 0; i-- )
64:                 m[i][n] -= m[i][k] * m[k][n];
65:         }
66:     }
-g .61
61:         for ( k = n-1; k >= 0; k-- ) {
-dl 10f0 1100
2B5F:10F0  00 00 00 00 00 00 00 40  +0.2E+1
2B5F:10F8  00 00 00 00 00 00 F0 3F  +0.1E+1
2B5F:1100  00 00 00 00 00 00 00 40  +0.2E+1
-dl 1148 1158
2B5F:1148  00 00 00 00 00 00 F0 3F  +0.1E+1   }
2B5F:1150  00 00 00 00 00 00 04 40  +0.25E+1  } 値がおかしい
2B5F:1158  00 00 00 00 00 00 00 40  +0.2E+1   }
-q
```

```
B>symdeb linear.sym linear.exe⏎ ……………プログラムを再度実行するために一度SYMDEB
Microsoft Symbolic Debug Utility          を終了してからもう一度起動する
Version 3.01                              (SYMDEB中で l コマンドで再ロードをすると
(C)Copyright Microsoft Corp 1984, 1985    シンボルファイルと実行ファイルを正しく参照
Processor is [80286]                      できないため)
-{linear.dat⏎
-s+⏎
-g _main⏎
linear.c
18:     {
-v .57⏎
57:                     for ( j = k+1; j <= n; j++ )
58:                         m[i][j] += a * m[k][j];
59:                 }
60:             }
61:             for ( k = n-1; k >= 0; k-- ) {
62:                 m[k][n] /= m[k][k];
63:                 for ( i = k-1; i >= 0; i-- )
64:                     m[i][n] -= m[i][k] * m[k][n];
-g .58⏎
+1 x0 +2 x1  = 1
+2 x0 +1 x1  = 2
58:                         m[i][j] += a * m[k][j];
-dl 1148 1158⏎
2B5F:1148  00 00 00 00 00 00 F0 3F  +0.1E+1
2B5F:1150  00 00 00 00 00 00 00 40  +0.2E+1
2B5F:1158  00 00 00 00 00 00 F0 3F  +0.1E+1
-p⏎ ……………………プログラムを1ステップ実行する
58:                         m[i][j] += a * m[k][j];
-dl 1148 1158⏎
2B5F:1148  00 00 00 00 00 00 F0 3F  +0.1E+1
2B5F:1150  00 00 00 00 00 00 04 40  +0.25E+1  } 値がおかしい
2B5F:1158  00 00 00 00 00 00 F0 3F  +0.1E+1
-p⏎
59:                 }                            ⇩
-dl 1148 1158⏎
2B5F:1148  00 00 00 00 00 00 F0 3F  +0.1E+1
2B5F:1150  00 00 00 00 00 00 04 40  +0.25E+1  } 58行目にバグがある
2B5F:1158  00 00 00 00 00 00 00 40  +0.2E+1
-q⏎

B>edlin linear.c⏎ ……………………edlinを使ってソースファイルの58行目を訂正
ファイルを終わりまで読込みました.
*58⏎
     58:*                   m[i][j] += a * m[k][j];
     58:*                   m[i][j] -= a * m[k][j];⏎
*e⏎

B>cl linear.c⏎
Microsoft (R) C Optimizing Compiler Version 5.10
Copyright (c) Microsoft Corp 1984, 1985, 1986, 1987, 1988. All rights reserved.

linear.c ⏎

Microsoft (R) Overlay Linker  Version 3.65
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.
```

```
Object Modules [.OBJ]: LINEAR.OBJ
Run File [LINEAR.EXE]: LINEAR.EXE /NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]:

B>type linear.dat⏎
  ........................方程式の次元
1.0  2.0  1.0········(ax0 + bx1 = c)を(a b c)と書く
2.0  1.0  2.0········(dx0 + ex1 = f)  (d e f)

B>linear < linear.dat⏎
+1 x0 +2 x1  = 1
+2 x0 +1 x1  = 2
x0 = 1
x1 = 0

B>
```

図7-11　SYMDEBによるデバッグ例

このようなデバッガのことを**シンボリックデバッガ**といいます．つまり，別にシンボル名とアドレスの対応データベースを用意しておき，それに基づいて，デバッグ時の画面表示をシンボルで行い，デバッグ作業を楽にするわけです．ただしこの場合，リンカなどのプログラムが，このシンボリックデバッガ用に対応していなければなりません．

■ UNIX上でのデバッグ　— adbの使い方 —

UNIXはマルチタスク・マルチユーザーのオペレーティングシステムなので，たとえばC言語でのポインタのまちがいなどによってオペレーティングシステム自体を壊してしまう，ということがないように作ってあります．

たとえば以下のようなプログラムがあった場合，MS-DOSではどんな動作をするのか，またプログラムを実行したあとにどんな影響を次のプログラムやオペレーティングシステム自身に与えているのかがわかりません．つまり，暴走しても，なにがなんだかわからない状態になることがあるのです．

このプログラムはコンパイルするとエラーもなくコンパイルができます．もちろん，MS-DOSでもUNIXでも同じようにコンパイルできます．

しかし，実行時にエラーが出るプログラムなのにもかかわらず，MS-DOSの場合，ときとしてエラーが出ないときがあります．このプログラムでは明確にポインタに「0」を入れていますから，こういった場合はエラーがライブラリによって検出されてプログラムが止まることが多いようですが，0でない不正な値が入った場合はその限りではありません．

UNIXやOS/2といったオペレーティングシステムでは，こういった実行時のバグが他のプロセスに影響を与えると，システムそのものやLANなどが止まってしまって，他のユーザーのみならず他の

マシンにも影響を及ぼすことになります．そこで，オペレーティングシステムはこういった不正なアクセスを常に監視していて，もし不正なアクセスがあると，そのプロセスを止めるようになっています．

ためしに**リスト 7-2** を UNIX 上でコンパイルして実行させ，エラーのある場所を UNIX の標準のデバッガである adb を使って探してみましょう．

```
 1: /* ***********************************************************
 2:     Test programs for Debugger
 3: *********************************************************** */
 4:
 5: #include <stdio.h>
 6: #include <ctype.h>
 7: #include <string.h>
 8:
 9: void    ErrorFunc() ……………………エラーのある関数
10: {
11:     char    *p;
12:
13:     p = (char *)NULL; ………………ポインタ変数にNULL(0)を代入し，その変数のさすアドレス(0)
14:     *p = 'A';                   に数値を代入する(当然エラーになる)
15: }
16:
17: void    NonErrorFunc() ……………エラーのない関数
18: {
19:     char    *p;
20:     char    q;
21:
22:     p = &q; ……………………………………こちらはエラーにならない
23:     *p = 'A';
24: }
25:
26: main()
27: {
28:     NonErrorFunc();     最初にエラーのない関数を呼び出し，次にエラーのある関数を呼び出す
29:     ErrorFunc();
30: }
```

リスト 7-2　tx.c(エラーのあるプログラム)

```
% ls -l
total 1
-rw-r--r--  1 nori          404 Sep 26 10:17 tx.c
        ┌──デバッガがかかる実行ファイルにする
% cc -g tx.c -o tx ……………コンパイル・リンクの実行(無事終了している)
                └──────実行ファイル名を指定する
% ls -l ……………ディレクトリを見て実行ファイルを確認
total 25
-rwxr-xr-x  1 nori        24576 Sep 26 10:22 tx*  ……………実行ファイル
-rw-r--r--  1 nori          404 Sep 26 10:17 tx.c ……………ソースファイル

% tx ……………実行
Segmentation fault (core dumped) ……………実行時にエラーの発生

% ls -l
total 89
-rw-r--r--  1 nori      2114362 Sep 26 10:22 core ……………デバッグ情報のファイル
-rwxr-xr-x  1 nori        24576 Sep 26 10:22 tx*
-rw-r--r--  1 nori          404 Sep 26 10:17 tx.c

% adb tx ……………デバッガを起動
core file = core -- program "tx"      }
SIGSEGV 11: segmentation violation    } デバッガのメッセージ
$c ……………「$c」と入力
_ErrorFunc() + 1a ……………ErrorFunc()という関数でバグがあって止まったことがわかる
_main(0x1,0xeffff04,0xeffff0c) + 10
(Control-D)
%
```

図7-12　tx.cのコンパイルと実行

adbはどのUNIXにも載っている標準のいちばん簡単なデバッガです．ここで示した使い方のほかにもたくさんの機能を持っていますが，詳しくはUNIXのライセンス元であるAT&T社の発行しているUNIXのマニュアルの「プログラマ・ガイド」を参照してください．

■ 高度なデバッガの問題点

最近は，ビジュアルな画面を持った使いやすいデバッガがコンパイラに付属してくるようになりました．

これらの高度な機能を持ったデバッガは非常に便利ですが，原始的なデバッガにもとからある欠点に加えて次のような欠点もあります．デバッガというのは，いわばプログラムをばらばらにして扱うという行為なので，これらの欠点を熟知した操作が必要です．さもないと，「実際のプログラムは動くが，デバッガ上では動かない」というようなおかしな現象に頭を悩ますことになります．

・Cコンパイラやリンカにデバッガ専用の情報を書き出す機能が備わっていなければならない．つまり，A社のデバッガはA社のコンパイラとA社のリンカ(それもA社のCコンパイラ専用リンカ)を使っていないとまったく動かない．また，これらのデバッグ情報のフォーマットが，Cコンパイラを供給しているメーカー同士でまちまちである．つまり，Cコンパイラとそのデバッグ環境は一体になっている．

・マウスなどの便利な外部デバイスが使えるデバッガは，そのデバイスに影響を与えるようなプログラムのデバッグはできない．

さて，こういった問題点を抱えている現在の高機能デバッガですが，やはりあるにこしたことはなく，便利な場面の方がむしろ多いでしょう．以上に述べた問題点を考えつつ行うデバッグならば，なんら問題はないということです．

索 引

A
abort関数 —— 274
accept関数 —— 281
adb —— 320
alarm関数 —— 274
archビット —— 131

B
bdos関数 —— 28, 123
bind関数 —— 281
brk関数 —— 271

C
CALL命令 —— 189
CLI命令 —— 201, 205
COMMAND.COM —— 145, 146
connect関数 —— 281

D
daemon —— 279
datagram —— 279
DEBUG —— 308
DGROUP —— 186, 201
dirビット —— 131
DTA領域 —— 126, 129

E
envp —— 211
EOI命令 —— 201
EUCコード —— 79, 81
execl関数 —— 142, 243
execle関数 —— 142, 243
execlp関数 —— 142, 243
execv関数 —— 142, 243
execve関数 —— 142, 243
execvp関数 —— 142, 243
exit関数 —— 241, 243

F
FAR —— 20, 189
FAR CALL —— 200
FINDFIRST —— 127
FINDNEXT —— 127
fork関数 —— 241, 243
free関数 —— 271

G
gethostbyname関数 —— 282
gethostname関数 —— 282
getprotobyname関数 —— 283
getservbyname関数 —— 283

H
hiddenビット —— 131
htonl関数 —— 282
htons関数 —— 282

I
IN命令 —— 195
inetdデーモン —— 283
inet_ntoa関数 —— 282
inp関数 —— 195
INT命令 —— 28, 124, 189, 313
int型変数 —— 13
intdos関数 —— 28, 123
intdosx関数 —— 28, 123
Internet-address —— 279
Internetドメイン —— 279
int86関数 —— 28, 124
int86x関数 —— 28, 124
I/Oアドレス —— 195
I/OマップドI/O方式 —— 194

J
JISコード —— 71

K
kernel —— 279
KI —— 88
kill関数 —— 274
KO —— 88

L
LIBコマンド —— 45, 48
listen関数 —— 281
long char型 —— 83, 100

M
MAKEFILE(MS-DOS) —— 300
Makefile(UNIX) —— 303
MAKEFLAGS —— 307
MAKEコマンド(MS-DOS) —— 300

makeコマンド(UNIX)——303
mallocl関数——271
MAPSYMコマンド——314
mkmfコマンド——307
msgbuf構造体——261
msgctl関数——262
msgget関数——261
msgqid_ds構造体——262
msgrcv関数——262
msgsnd関数——261

N
Named Pipe——254
NEAR——20, 189
ntohl関数——282
ntohs関数——282

O
outp関数——195
OUT命令——195

P
PATH——146
pause関数——274
pclose関数——256
pipe関数——255
popen関数——256
printf関数——21
publicシンボル——46
PUSH命令——186

R
ranlibコマンド——49
realloc関数——271
recvfrom関数——281
register変数——186
REGS共用体——30, 122
RETF命令——186
R/Oビット——131

S
sbrk関数——271
segread関数——125
select関数——281
sembuf構造体——248
semctl関数——249
semget関数——248
semid_ds構造体——250
seminfo構造体——249
semop関数——248
sendto関数——282
SETDTA——127
sethostname関数——282
shmat関数——269
shmctl関数——269
shmdt関数——269
shmget関数——269
shmid_ds構造体——269
signal関数——273
socket関数——281
spawnl関数——142, 144
spawnle関数——142, 144
spawnlp関数——142, 144
spawnv関数——142, 144
spawnve関数——142, 144
spawnvp関数——142, 144
SREGS構造体——30, 122
STI命令——201
strchr関数——98
strlwr関数——99
strpbrk関数——98
swapper——240
SYMDEB——309, 314
syslog関数——284
system関数——143, 145
systemビット——131

U
ulimit関数——271

V
virtual-circuit——279
volumeビット——131

W
wait関数——241, 243

ア
アセンブラプログラム——184
アタッチ——268
インタラプトベクタ——197
インライン・アセンブル——188
エスケープシーケンス——65, 210
親プロセス——141, 241

カ
階層ディレクトリ——149
カーネル——279
拡張UNIXコード——79
画面制御——87, 210
環境変数——144, 211
漢字アウト——88
漢字イン——88
漢字コード——71
キー番号——246
キャラクタデバイス——135

旧JIS ― 74
共有資源 ― 239
共有メモリ ― 268
禁則 ― 104
クライアント／サーバーモデル ― 279
クロック周波数 ― 205
コアイメージ ― 272
交通信号機型セマフォ ― 247
コード領域 ― 18
子プロセス ― 141, 241
コンパイル ― 185, 297
コンパクトモデル ― 19

サ

サーバー ― 280
シグナル ― 272
シグナル番号 ― 273
資源の共有 ― 237
システムコール ― 27, 117, 239
シフトJISコード ― 75, 93
シフトコード ― 71
時分割 ― 237
シリアルコントローラ ― 206
新JIS ― 74
シングルタスク ― 237
シンボリックデバッガ ― 311, 320
シンボルファイル ― 314
スタック領域 ― 18
スモールモデル ― 19, 123
制御端末 ― 284
制御文字 ― 88
セグメント ― 16, 186
セグメント空間 ― 18
セマフォ ― 246
全角文字 ― 73, 86, 91
ソケット ― 279
ソフトウェア割り込み ― 189

タ

タイニィモデル ― 26
タイムスタンプ ― 129, 151, 302
タスク ― 189
ターミナルエミュレータ ― 183, 190
ディレクションフラグ ― 201
ディレクトリ ― 126
ディレクトリエントリ ― 149
データグラム ― 279
データセグメント ― 271
データ領域 ― 18
デバイスファイル ― 254
デーモン ― 279, 283
ドメイン ― 279

ナ

名前つきパイプ ― 254
入場制限型セマフォ ― 247

ハ

排他制御 ― 247
バイナリサーチ ― 94
パイプ ― 254
バーチャルサーキット ― 279
バッファ ― 204
バッファリング ― 87
ハードウェア割り込み ― 189
パラレルインタフェース ― 207
半角文字 ― 76
ビットフィールド ― 13, 130
ヒュージモデル ― 20
ファイル属性 ― 131
ファイルディスクリプタ ― 254, 279
ファンクションコール ― 121
物理アドレス ― 16
ブレークバリュー ― 271
ブレークポイント ― 313
プログラマブル・カウンタ ― 195
プログラム開発ツール ― 40
フロー制御 ― 212
プロセス ― 240
プロセス間通信 ― 239, 254, 268
プロセス管理 ― 241
プロセス制御 ― 246
ブロックサイズ ― 151
ポインタ演算 ― 18
ポインタ長 ― 20
ポインタ変数 ― 14
ポート番号 ― 279, 286
ボリュームラベル ― 131, 135

マ

マクロ ― 305
マクロ機能 ― 65
マクロ・ライブラリ ― 65
マップファイル ― 46, 310, 314
マルチタスク ― 237, 239
マルチユーザー ― 237
ミディアムモデル ― 19
メッセージ ― 260
メモリ管理 ― 270
メモリマップ ― 21
メモリマップドI/O方式 ― 194
メモリモデル ― 18, 26

ラ

ライブラリ ― 43
ライブラリ関数 ― 64

ライブラリ管理 ―― 48
ライブラリ・マネージャ ―― 45
ラージモデル ―― 14, 19
ラベル ―― 186, 306
リソース ―― 237, 247
リダイレクト ―― 145
リターンコード ―― 146
リンク ―― 43, 184, 297
リングバッファ ―― 212
レジスタ構造体 ―― 28
レジスタ変数 ―― 15
論理アドレス ―― 16

ワ

ワイルドカード ―― 127, 150
割り込みコントローラ ―― 199
割り込みサービスルーチン ―― 191
割り込みサポートライブラリ ―― 51
割り込み処理 ―― 189, 272
割り込みベクタ ―― 124
割り込みベクタアドレス ―― 207
割り込みマスクレジスタ ―― 207

【参考文献】

■ C 言語関連

- 「ANSI X3J11/89-xx」
- 「ANSI-C A Lexical Guide」　Mark Williams Cmpany, PRENTICE HALL
  ISBN0-13-037814-3
- 「ANSI C 言語辞典」　平林雅英著，技術評論社
  ISBN4-87408-320-X
- 「詳説 C 言語(H&S リファレンス)」
  サミュエル・P・ハービソン／ガイ・L・スティール共著，斉藤信男監訳，日本ソフトバンク
  ISBN4-89052-050-3
- 「プログラミング言語 C　第 2 版」
  ブライアン・W・カーニハン／デニス・M・リッチー共著，石田晴久訳，共立出版
  ISBN4-320-02483-4

■ アセンブラ関連

- 「はじめて読むアセンブラ」　村瀬康治著，アスキー出版局
  ISBN4-87148-774-1
- 「はじめて読む 8086」　蒲地輝尚著，村瀬康治監修，アスキー出版局
  ISBN4-87148-245-6

■ UNIX 関連

- 「実用 UNIX システムプログラミング」　塩谷修著，日刊工業新聞社
  ISBN4-526-02054-0
- 「UNIX システムコール・プログラミング」　Marc J.Rochkind 著，福崎俊博訳，アスキー出版局
  ISBN4-87148-260-X

【筆者紹介】

三田 典玄（みた のりひろ）

1957年生れ.
オーディオ関連会社，コンピュータ・エンジニアを経て，1986年2月Coredump Co.,Ltdを設立．現在は，その代表取締役社長として国内のみならず海外までも各所を飛び回り超多忙な日々を送っている.
また，UNIXマシンを個人で所有し，パソコン・ネットワーク(Personal Unix Net)を開設中．UNIXに関しても造詣が深い.

■編集協力
西　克弘

応用C言語　改訂新版
アスキー・ラーニングシステム③応用コース
1988年2月21日　初版発行
1990年12月21日　第2版第1刷発行
1994年5月21日　第2版第11刷発行

著　者　三田 典玄（みた のりひろ）
発行人　宮崎 秀規
編集人　佐藤 英一
発行所　株式会社アスキー
〒151-24　東京都渋谷区代々木4-33-10
振　替　東京4-161144
大代表　(03)5351-8111
出版営業部 (03)5351-8194（ダイヤルイン）
第一書籍編集部 (03)5351-8106（ダイヤルイン）

制　作　株式会社 GARO
印　刷　株式会社 加藤文明社

ISBN4-7561-0056-2　　Printed in Japan

●11214／11405／11566